Hybster

A Highly Parallelizable Protocol for Hybrid Fault-Tolerant Service Replication

Johannes Behl
TU Braunschweig
behl@ibr.cs.tu-bs.de

Tobias Distler
FAU Erlangen-Nürnberg
distler@cs.fau.de

Rüdiger Kapitza
TU Braunschweig
rrkapitz@ibr.cs.tu-bs.de

April 21st, 2017
v0.1

Contents

1 Introduction

Hybster is a replication protocol for providing fault-tolerant services based on the state-machine approach [13]. It is designed for partially synchronous systems [9]. It does not rely on timing assumptions to ensure safety but is only guaranteed to make progress in phases where unknown upper bounds for message delivery and message processing exist. Moreover, using a hybrid fault model in which some components are trusted to only fail by crashing even when other components behave arbitrarily faulty [12, 14], Hybster requires only $2f + 1$ service replicas to tolerate up to f faulty ones.

While the original paper that introduces Hybster [3] mainly presents the rationale behind the basic protocol and its parallelized variant, this technical report provides supplemental material such as a comprehensive system model description and formal specification of the protocol. The present document is the first version of the report that is planned to be updated once additional parts are ready. All published versions will be accessible at [2].

2 Notation

The formal specification of Hybster is essentially based on I/O automata [11] as employed by Castro and Liskov to specify the originator of Byzantine fault-tolerant service replication, namely PBFT [5, 6, 8]. However, the model used here differs from I/O automata mainly in two aspects. Firstly, it supports the direct interaction of modules where module instances explicitly trigger operations on other module instances. This makes dependencies between modules more apparent than the indirect interaction scheme based on operation signatures as used by the original I/O automata. Moreover, it allows more flexible interaction patterns between modules, especially helpful for describing the internal structure of complex processes. Secondly, to improve readability, the notation used in this document is inspired by pseudo-code notations like [4].

2.1 Modules and Operations

The specification of Hybster is given as a set of *modules* encapsulating *state* and defining *operations* on that state. Compared to an I/O automaton, a module is a more abstract concept. A module is not only a concurrent entity but allows for both asynchronous and synchronous operations. Furthermore, modules can be instantiated, there can be multiple identifiable *instances* of one and the same module. Instances share the behavior as described by the operations of the module but possess independent memory for state variables. Thus, modules and module instances have strong similarities to classes and objects of the object-oriented programming paradigm [1].

Tasks and Methods. Asynchronous operations are called *tasks* and are triggered according to a specified precondition. If the precondition of a task is satisfied for a module instance, the task of this instance is *enabled* and all enabled tasks are assumed to be executed asynchronously at some time, in a nondeterministic order, although only one task at a time. Tasks are classified as input, output, or internal operation. While tasks are not allowed to yield a direct return value, output tasks may specify a list of output parameters that serve as inputs for other tasks. Likewise, input tasks define a list of expected input parameters. Internal tasks can possess a parameter list as well. These parameters are neither inputs nor outputs and their only purpose is to make the function, the logical result of the internal tasks explicit.

Synchronous operations, called *methods*, are directly executed when invoked and may return a direct outcome. Methods are also classified as input, output, and internal operation but can be additionally declared as public. A public method is permitted to alter the state of the invoked module instance and yield a direct result; a public method can be regarded as both input and output. Besides tasks and methods, *global functions* can be specified. Such operations are not bound to instances of modules thus they cannot access any instance state.

Figure 1: Example for module definitions.

```
 1 module NumberProducer_p
 2   upon init() do
 3     out_p ∈ (ℕ)* := []

 5   upon internal task produce(val ∈ ℕ)
 6     with
 7       |out_p| < 10
 8       val = random(0, 100)
 9     do
10       out_p.append(val)

12 module NumberConsumer_c
13   upon init() do
14     sum_c ∈ ℕ := 0

16   upon internal call consume(val ∈ ℕ) do
17     sum_c := sum_c + val
```

Figure 1 gives an example for the definition of two modules. The module *NumberProducer* generates random numbers within an internal task *produce* and *NumberConsumer* summarizes numbers within an internal method *consume*. Both modules are not yet connected. How they can interact is described in the next section. The freely chosen index p in "**module** NumberProducer$_p$" identifies an instance of the *NumberProducer* in the context of the module definition. The special method *init* is used to initialize a newly created instance of a module. "$out_p \in (\mathbb{N})^* := []$" in Line 3 defines a single variable *out* for a new instance of the module *NumberProducer* denoted with p. out_p is an instance ($\in$) of a list ($(\dots)^*$) of natural numbers ($\mathbb{N}$) that is initially assigned ($:=$) to an empty list ($[]$). The definition of the *produce* task starts in Line 5. It is not allowed to have a direct return value. However, the parameter *val* is a free variable that is eventually bound to the value *produce* generates. *val* cannot be used directly. At this point, its only purpose is to make the function of *produce*, its logical outcome explicit. The **with** keyword starts the precondition section of the task definition. *produce* is only enabled when the list out_p contains fewer then 10 elements. Since *val* is a free variable, binding *val* to some randomly chosen natural number by "$val = random(0, 100)$" is always satisfied. In this example, *random* is assumed to be a global function defined within some other module. When a task is enabled, eventually all imperative statements in the section introduced by the keyword **do** will be executed in the sequence as specified. *produce* comprises only a single statement. "$out_p.append(val)$" adds the value bound to the variable *val* to the end of the list referenced by out_p. In sum, *produce* appends randomly chosen numbers to the list out_p as long as out_p currently contains fewer than 10 elements. The definition of *NumberConsumer* starting in Line 12 uses c to refer to an instance of that module. Instances of *NumberConsumer* possess one member variable sum_c that is a natural number initialized with 0. Further they possess one internal method *consume* that takes a value as argument and adds the value to sum_c.

Operation Definition and Declaration. The complete syntax for the definition of operations can be found in Figure 2. The keyword **upon** begins the definition of a task or a method. It is followed by an access modifier and an optional declaration of the operation type (**task** for tasks and **call** for methods). Operations may define a list of parameters. In the case of output and internal tasks, the parameters are output, in all other cases input parameters. Methods

Figure 2: Syntax for operation definitions.

(a) Define tasks.

```
upon <internal|input|output> [task]
    taskName(arg_0 ∈ T_0, ..., arg_{n-1} ∈ T_{n-1})
  asserts
    // assertions
  with
    // preconditions
    assert O() // included assertion
    [if] P()
  do
    // statements
  yields
    // postconditions
```

(b) Define methods and global functions.

```
upon <internal|input|output|public|global>
    [call] methodName(arg_0 ∈ T_0, ...) → ret ∈ R
  asserts
    // assertions
  do
    // statements
    return ret
  yields
    // postconditions
```

may additionally define a direct return parameter, although this is optional. Access modifier, type declaration, name, and the list of all parameters form the *signature* of an operation. It follows a number of sections that constitute the body of the operation definition. *Assertions* are predicates that are expected to be true, they are assumptions that shall be made explicit. Assertions can be declared within the **asserts** section or anywhere else following an **assert**. A task must specify a set of *preconditions* that determine when the task is enabled and thus eligible for execution. Preconditions are given in the **with** section. Single preconditions can be prefaced by an optional **if** and if multiple conditions are specified, they are implicitly combined by a logical AND operation ($\wedge$). If an enabled task is executed or if a method is invoked, the sequence of *statements* within the **do** section is evaluated. Unless the operation is a global function, statements are allowed to modify the state of the invoked module instance. If a **return** statement is reached, the evaluation of the operation ends. For methods with return parameter, the **return** keyword has to be followed by an expression that specifies the return value. In conjunction with or as replacement for statements that explicitly alter state or define the values of output and return parameters, a set of *postconditions* may be declared that have to be satisfied after the evaluation of an operation. Postconditions are located in the **yields** section.

Figure 3: Syntax for operation declarations.

(a) Declare tasks.

```
declare <internal|input|output> task
    taskName(arg_0 ∈ T_0, ..., arg_{n-1} ∈ T_{n-1})
```

(b) Declare methods and global functions.

```
declare <internal|input|output|public|global>
    call methodName(arg_0 ∈ T_0, ...) → ret ∈ R
```

Operation definitions are both a declaration and a direct implementation. If just a type or a part of an interface is to be given, the syntax shown in Figure 3 can be used to merely declare operations. The implementation has to be provided by a different module.

2.2 Module Interaction

Modules are not only used to describe a single algorithm, they also serve as interacting components in the specification of more complex systems. To describe various patterns of how modules can interact, the modules *NumberProducer* and *NumberConsumer* defined in Figure 1 of the previous section are extended in different ways. In contrast to the *indirect interaction* scheme of I/O automata in which output operations of one automaton are linked to input operations

Figure 4: Direct unidirectional module interaction.

(a) Source-linked.

```
1  module LinkingNumberSource_p
2    extends NumberProducer

4    upon init() do
5      snk_p ≔ NumberSink()

7    upon internal task transmit()
8      with
9        |out_p| > 0
10     do
11       output snk_p.receive(out_p.dequeue())

13 module NumberSink_c
14   extends NumberConsumer

16   upon input call receive(val ∈ ℕ) do
17     consume(val)
```

(b) Sink-linked.

```
1  module NumberSource_p
2    extends NumberProducer

4    upon output task transmit(val ∈ ℕ)
5      with
6        |out_p| > 0
7      do
8        val ≔ out_p.dequeue()

10 module LinkingNumberSink_c
11   extends NumberConsumer

13   upon init() do
14     src_c ≔ NumberSource()

16   upon internal task receive(val ∈ ℕ)
17     with input src_c.transmit(val) do
18       consume(val)
```

of other automata on the basis of the operations' signature [11], in the model employed here, instances of modules invoke operations of other instances *directly*.

Unidirectional Interaction. For example, Figure 4 shows two forms of a *direct unidirectional interaction*. Generally, in an unidirectional interaction, the output of one module, the source, is taken as input for another, the sink. Figure 4a presents the *source-linked* variant of this interaction scheme. In this example, two modules are defined, *LinkingNumberSource* and *NumberSink*. The module *LinkingNumberSource* is derived from the module *NumberProducer* and the current instance of this module is referred to as *p*. *LinkingNumberSource* inherits from *NumberProducer* the instance variable out_p maintaining the list of generated but not yet consumed random numbers and the internal task *produce*. Additionally, it defines the instance variable snk_p initialized with a new instance of the module *NumberSink* and an internal task *transmit*. *transmit* is enabled when out_p contains at least one generated number. Upon being executed, *transmit* removes the first value from the list out_p ($out_p.dequeue()$) and invokes the input method *receive* of the instance snk_p with this value as argument. The method invocation is synchronous, that is, all of its effects take place immediately. While being optional, the preceding keyword **output** shall emphasize this particular interaction between *LinkingNumberSource* and *NumberSink*. The module *NumberSink* is derived from *NumberConsumer* and thus inherits the variable sum_c and the internal method *consume*, with *c* denoting the current module instance. The input method *receive* of *NumberSink* simply invokes *consume* of its base module handing over the given value *val* as argument. Taken all together, the task *produce* defined by *NumberProducer* asynchronously generates random natural numbers and places them in the list out_p as long as this list contains fewer than 10 values. The task *transmit* asynchronously removes values from this list and delivers them as input to the *NumberSink* instance snk_p. The invoked input method *receive* of snk_p calls the internal method *consume* that finally adds the given value to the instance variable sum_c, with *c* being the internal reference of the module instance also referenced by snk_p.

Taken a source-linked interaction, the source is bound to the type of the sink and the source

instance needs a reference to the instance of the sink. In a *sink-linked* interaction scheme, this relation is reversed. The example depicted in Figure 4b resembles the previous one. The module *NumberSource* extends *NumberProducer* and defines the operation *transmit* whereas the module *LinkingNumberSink* extends *NumberConsumer* and defines the operation *receive*. However, in this case, *transmit* is an output task that delivers the next random value from the list out_p by means of the output parameter *val*. The addressee of this value is not known to the *NumberSource* module. Instead, instances of *LinkingNumberSink* maintain a reference src_c to a *NumberSource* instance and their internal *receive* task is bound to $src_c.transmit$, that is, *receive* is directly called after $src_c.transmit$ has been executed. Analogous to the **output** keyword, the **input** keyword emphasizing the interaction between *NumberSource* and *LinkingNumberSink* may be omitted. While receiving operations in sink-linked interactions are allowed to wait for multiple output events from different instances or even different output operations, they must not specify additional preconditions that would prevent any of their awaited outputs from being executed. In other words, sink-linked interactions must remain input-enabled as required from I/O automata [11].

Bidirectional Interaction. Whereas unidirectional interaction schemes link outputs to inputs, *method invocations* are a form of *bidirectional interaction* in which a callee takes an input and directly returns an output. Figure 5a presents an example. Again, *NumberProducer* and *NumberConsumer* from Figure 1 are extended to create interacting modules. The resulting modules *IncreasingSum* and *Accumulator* resemble the modules *LinkingNumberSource* and *NumberSink* of the source-linked interaction example presented in Figure 4a. Opposed to the input method *receive* of *NumberSink*, the public method *add* of *Accumulator* does not only add the given value to the variable sum_c but also returns the resulting sum as direct outcome of the method invocation. To emphasize the invocation at the caller's side, the keyword **invoke** can be used.

Figure 5: Further module interaction schemes.

(a) Method invocation.

```
1  module IncreasingSum_p
2    extends NumberProducer

4    upon init() do
5      acc_p ≔ Accumulator()
6      sum_p ∈ ℕ ≔ 0

8    upon internal task transmit()
9      with
10       |out_p| > 0
11     do
12       sum_p ≔ invoke acc_p.add(out_p.dequeue())

14 module Accumulator(NumberConsumer)_c
15   upon public call add(val ∈ ℕ) → sum ∈ ℕ do
16     consume(val)
17     return sum_c
```

(b) Global function.

```
1  module Computer_c
2    upon global call thinkDeep() → ℕ do
3      return 42

5  module Being_b
6    upon init() do
7      answer_b ∈ ℕ ∪ {∅} ≔ ∅

9    upon internal task askUltimate()
10     with
11       answer_b = ∅
12     do
13       answer_b ≔ Computer::thinkDeep()
```

Global Functions. *Global functions* share the syntax with methods but are not bound to any instance. While being specified within some module, they can be called from anywhere. Fig-

ure 5b illustrates the usage. The function *thinkDeep* is defined within the module *Computer* and simply returns 42 as constant. It is called by the module *Being* via "*Computer*::*thinkDeep*()". Stating the defining module explicitly is, however, optional.

Inner Modules. Another form of interaction that leads to a very tight coupling between modules is the use of *inner modules*. An instance of an inner module possesses an implicit reference to the module instance that creates it and has access to all instance variables and operations be it internal or external of the referenced outer module instance. Figure 6 gives an example. Instances of the module $OuterNumberSource$ create an instance of the inner module $InnerNumberSink$. The inner module instance thereby gets access to the list out_c of produced random numbers that $OuterNumberSource$ inherits from $NumberProducer$. If out_c contains values, the internal task *consumeInner* of $InnerNumberSink$ is enabled. When executed, the inner instance removes a value from out_c maintained in the outer instance and passes it as argument to the *consume* method $InnerNumberSink$ inherits from $NumberConsumer$.

Figure 6: Inner modules.

```
1  module OuterNumberSource_p
2    extends NumberProducer

4    upon init() do
5      snk_p := OuterNumberSource_p::InnerNumberSink()

7  module OuterNumberSource::InnerNumberSink_c
8    extends NumberConsumer

10   upon internal task consumeInner(val ∈ ℕ)
11     with
12       |out_c| > 0
13     do
14       consume(out_c.dequeue())
```

2.3 Additional Expressions and Types

While modules and their various ways of interaction are related to the general architecture of a system, the purpose of operations is to specify concrete algorithms by declaring pre- and postconditions or by defining them step-by-step as sequence of actions. This section gives an overview of expressions and basic types used to form predicates and imperative statements for the definition of operations.

Sets. Sets as unordered collections of objects are denoted by curly brackets. For example, $\{1, 2, 3\}$ defines the set of the natural numbers 1, 2, and 3. The empty set can be written as $\{\}$ or $\emptyset$. Concrete sets are instances of the basic type *Set*. $X \in Set$ declares an arbitrary set X, and $X := \{1, 2, 3\}$ assigns the set $\{1, 2, 3\}$ to it. Common set operations are determining the union ($\cup$) or the difference ($\setminus$) of two sets. For example, $X := X \setminus \{2\} \cup \{4\}$ results in $X = \{1, 3, 4\}$. The power set of some set Y is given by $\mathcal{P}(Y)$. This can be used to specify a typed set: $Z \in \mathcal{P}(\mathbb{N})$ declares Z as a set of natural numbers.

Tuples. The ordered counterparts of sets are tuples which are instances of the basic type *Tuple*. Given n objects x_0 to x_{n-1}, the corresponding tuple is written as $\langle x_0, \ldots, x_{n-1} \rangle$. For

example, $t \in Tuple \coloneqq \langle 1, 2, 3 \rangle$ defines t as the tuple comprising the elements 1, 2, and 3 in that order. t could have been also declared as $t \in \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ or $t \in \mathbb{N}^3$, additionally specifying the types of the tuple's elements.

Lists. While tuples are immutable, lists are ordered collections that can be modified. Lists are instances of the basic type *List* that is also denoted by $(\dots)^*$. For example, a list l of natural numbers can be declared with $l \in (\mathbb{N})^*$. Lists are given within square brackets. $l \coloneqq [1, 2, 3]$ initializes l with the elements 1, 2, and 3. The size of l is $|l| = 3$. The type *List* defines the methods *append*, *pop*, and *dequeue*. *append* adds an object to the end of the list, *pop* returns and removes the last element and *dequeue* the first element. Taken the list l for example, $l.append(4)$ results in $l = [1, 2, 3, 4]$, $x \coloneqq l.pop()$ in $x = 3 \land l = [1, 2]$, and $x \coloneqq l.dequeue()$ in $x = 1 \land l = [2, 3]$.

Booleans. Classical truth values are represented by the basic type $Bool \overset{\text{def}}{\equiv} \{True, False\}$, with $True \overset{\text{def}}{\equiv} \emptyset = \emptyset$ and $False \overset{\text{def}}{\equiv} \emptyset \neq \emptyset$. Based on that, a predicate can be defined as a method or function returning a value of type *Bool*: **upon** $[\dots]$ **call** $predicateName(\dots) \to Bool$. An alternative notation for $True$ is $\top$ and for $False$ $\bot$.

Ellipses. Predicates may take several input parameters. Sometimes it is sufficient to verify that a predicate holds for a subset of these parameters in conjunction with any binding of the remaining ones. For example, given a ternary predicate P and two particular arguments x and y, the question could be if P is fulfilled for x and y and any additional argument z. To simplify the resulting expression $\exists z P(x, y, z)$, the elliptic form $P(x, y, \cdot)$ can be used. The extended variant for multiple arbitrarily chosen input parameters is written as $P(x, \dots) \overset{\text{def}}{\equiv} \exists y \exists z P(x, y, z)$.

None. At some occasions it can be helpful to bind a variable to a special value that resides outside the main domain of that variable, for instance, to denote that the variable is unset and does not contain any real value. As such special value *None* may be used, also denoted by $\varnothing$. With $x \in \mathbb{N} \cup \{\varnothing\}$, the variable x can reference a natural number or the special value $\varnothing$.

In addition to basic types, specifications can make use of common control flow structures such as conditionals and loops but also of more specific structures with regard to asynchronous operations.

Sequences of Statements. Sequences of statements are usually given as list, each statement in a single line at the same level of indentation. Figure 7a shows an example in which the values of two variables x and y are swapped using an additional variable *tmp*. An alternative notation of this example where statements are written in a single line separated by semicolons is given in Figure 7b. If single statements are wrapped, be it for clarity or because they do not fit into the space for a single line, overhanging parts have to be indented by two levels. As Figure 7c illustrates, this can happen within a single line or by splitting the statement such that it spans multiple lines.

Conditionals and Loops. The notation also supports typical control flow statements for conditional branches or repetitive executions. Figure 8a gives an example in which two variable x and y are compared and a return value is chosen depending on the outcome of this comparison. If a condition is satisfied, the associated block indented by one level is executed. Although these

Figure 7: Sequences of statements.

(a) Multiple lines.

```
1 tmp ≔ x
2 x ≔ y
3 y ≔ tmp
```

(b) Single line.

```
1 tmp ≔ x; x ≔ y; y ≔ tmp
```

(c) Long lines.

```
1 lx ≔ "This is a long expression
      written in a single line."
2 ly ≔ "And this is another long
3       expression but written
4       using multiple lines."
```

blocks contain only a single **return** in this example, blocks can generally span an arbitrary number of statements. An example for a loop that iterates over all elements of a set X and passes them to a procedure *processX* is depicted in Figure 8b.

Figure 8: Control flow statements.

(a) Conditionals.

```
1 if x < y then
2   return x
3 else if y < x then
4   return y
5 else
6   return 0
```

(b) Loops.

```
1 for all x ∈ X do
2   processX(x)
```

Synchronous Task Evaluations. As described in Section 2.1, tasks are enabled when their specified precondition is satisfied but they are only executed asynchronously without any guaranteed order. Furthermore, if the precondition of a task t depends on state that is altered by other tasks such that t is enabled and disabled depending on which other tasks are executed, there is no guarantee that the task t is executed at all. To support occasions in which a task is to be executed synchronously if necessary, the **check** keyword is provided. For example, if "**check** *someTask*()" is called, the precondition of task *someTask* is evaluated and, if the precondition is satisfied, *someTask* is executed like a synchronous operation. Thus, the task is transformed into a conditional method invocation.

3 System Model

The system model Hybster is based on is similar to the one used by PBFT [5, p. 26ff.]. Briefly, in a partially synchronous system with unreliable network, clients invoke operations of a stateful service that is replicated across several servers called replicas in order to ensure that the service does not return incorrect results even if a bounded number of these replicas behaves arbitrarily faulty. As the most notable difference to PBFT, however, Hybster also assumes that processes can be equipped with subsystems that are more reliable than other components such that these subsystems never fail in an arbitrary manner but only by crashing. Assuming such trusted subsystems, Hybster requires fewer replicas than PBFT to tolerate the same number of faults. A more detailed description of Hybster's system model is given in the following.

3.1 Processes

Hybster is designed for a distributed system in which processes communicate exclusively via message passing over a shared network. Two types of processes are distinguished, *replicas* and

clients. A set of replicas, also called *replica group*, provides a replicated realization of a service, with each replica hosting an instance of an implementation of that service. Clients invoke operations of the service by means of a local invocation handler instance encapsulating the communication with the replicas. The set of replicas in the system $\{r_0, r_1, \ldots, r_{n-1}\}$ is denoted by $\mathcal{R}$ and the number of replicas by n, that is, $n \stackrel{\text{def}}{\equiv} |\mathcal{R}|$. Analogously, the set and numbers of clients is given by $\mathcal{C} \stackrel{\text{def}}{\equiv} \{c_0, c_1, \ldots, c_{nclients-1}\}$ and $nclients \stackrel{\text{def}}{\equiv} |\mathcal{C}|$.

Listing 1: Specification of a deterministic stateful service implementation.

```
1  module Service_s

3  upon init() do
4    state_s ∈ S := s_0

6  upon public call invoke(c ∈ C, invno ∈ ^iN, svccmd ∈ O) → O' do
7    s', svcret := calculateResult(state_s, c, invno, svccmd)
8    state_s := s'
9    return svcret

11 declare internal call calculateResult(s ∈ S, c ∈ C, invno ∈ ^iN, svccmd ∈ O) → S × O'

13 upon output call createStateSnapshot() → S do
14   return state_s

16 upon input call installStateSnapshot(s ∈ S) do
17   state_s := s
```

The replicated service is assumed to be stateful and its implementation realized as a deterministic state machine. That is, starting from the same state und provided with the same command as input, the service implementation, if correct, always yields the same result independent of the instance and independent of the replica that executes it. Listing 1 shows the detailed specification of the service implementation. All possible states of the service's deterministic state machine are represented in the set $\mathcal{S}$. Each instance of the type *Service* starts in the same initial state $s_0 \in \mathcal{S}$ that is assigned to the instance variable $state_s$. The main interface of a service instance is its public *invoke* method. *invoke* expects a service command *svccmd* of the type $\mathcal{O}$ that encapsulates the invoked service operation together with all required arguments. Moreover, *invoke* has to be provided with the identifier $c \in \mathcal{C}$ of the client that issued the command and a sequence number *invno* assigned by the client to identify the particular invocation. To distinguish different types of sequence numbers, invocation numbers are of the type ${}^i\mathbb{N} \stackrel{\text{def}}{\equiv} \mathbb{N}$. The method *invoke* does not implement the state transition itself. For that purpose, it relies on the internal method *calculateResult* that deterministically maps a specified current state and the invocation information comprising the client, its invocation number, and the service command to a new state and a return value of the type $\mathcal{O}'$. Being dependent of the concrete service, the method *calculateResult* is not implemented but only declared. *invoke* calls *calculateResult* on the basis of the state referred to by the instance variable $state_s$. When the call returns, it sets $state_s$ to the new state and returns with the return value calculated by the command execution. Besides the main method for invoking operations of the realized service, the type *Service* offers methods to create and install snapshots of the service state.

The interface of the invocation handler that is used by clients to issue commands to the replicated service is depicted in Listing 2. The method *startInvocation* takes a service command

Listing 2: Specification of the invocation handler interface.

```
1 module InvocationHandler_c
2   declare input call startInvocation(svccmd ∈ O)
3   declare output task invocationCompleted(svcret ∈ O')
```

of the type $\mathcal{O}$ as input. The respective return value of the command execution is eventually delivered by the output task *invocationCompleted*. Correct clients are only allowed to start a new invocation when a previous invocation was completed, that is, when no other invocation is in progress.

3.2 Time

The system is assumed to be partially synchronous [9]. More precisely, it is assumed that there exist upper bounds for the delivery of messages and how long it takes a process to carry out particular operations but that these upper bounds are not known a priori.

Listing 3: Specification of timers with individual clock speeds.

```
1 module Timer_s

3 upon init() do
4   timeout_s ∈ ℕ ∪ {∅} := ∅

6 upon input call schedule(timeout ∈ ℕ) do
7   timeout_s := timeout

9 upon internal task tick()
10  with
11    timeout_s ≠ ∅ ∧ timeout_s > 0
12  do
13    timeout_s := timeout_s − 1

15 upon output task timerExpired()
16  with
17    timeout_s ≠ ∅ ∧ timeout_s = 0
18  do
19    timeout_s := ∅

21 upon output call isScheduled() → Bool do
22  return timeout_s ≠ ∅

24 upon input call cancel() do
25  timeout_s := ∅
```

In this model, processes have access to local timers that advance at individual speeds. An upper bound for the relative speeds exists but is unknown. Listing 3 presents the specification of a type *Timer* that can signal when a configured amount of time that is relative to each *Timer* instance elapsed. For that purpose, each instance maintains a variable $timeout_s$ that stores the remaining number of clock ticks until the timer expires or *None* if no timeout is currently configured. The input method *schedule* can be used to start the timer. It sets $timeout_s$ to the specified number of clock ticks. The internal task *tick* regularly decrements the current value of $timeout_s$ by one as long as $timeout_s$ is greater then zero. Though, it depends on the relative

speed of the timer instance in which interval the task is executed. Once the value of $timeout_s$ reaches zero, the output task *timerExpired* signals that the previously configured number of ticks has elapsed. In addition to the basic functionality, the type $Timer$ defines methods to determine if a timeout is currently configured and to stop the timer before *timerExpired* is triggered.

3.3 Communication

The network used by processes to communicate with each other by sending messages can be unreliable. It is allowed to reorder, duplicate, delay, and even drop messages. As a consequence, the assumed upper bound for message delivery only holds if a message is sent sufficiently often. Since the upper bound is not known a priori, it is not known in advance how often this is.

The specification of the network and its conditions can be found in Listing 4. Messages (Listing 4a) are modeled as tuples that contain at least one element identifying the message type. A message of the type Msg can be written as $\langle \text{Msg}, [\ldots] \rangle$, where $[\ldots]$ is a placeholder for a list of optional elements. From the network's perspective, processes act as sinks that receive any kind of message (Listing 4b). The specification of the network itself (Listing 4c) is based on the one given by Castro [5, p. 28]. The state of a network comprises the set of all connected processes stored in $nodes_n$ and the set $wire_n$ that maintains a pair for all messages that are currently in transmission. The first element of such a pair is the message in question and the second is a set of processes supposed to receive the message. To request the network to transmit a message m, the input method *send* is provided. It takes m as argument together with a subset R of the connected processes and notes R in $wire_n$ as intended recipients for m. Delivering messages is the responsibility of the internal task *transmit*. It removes a recipient r of an existing pair $\langle m, R \rangle$ in $wire_n$ and invokes the *receive* method of r passing m as argument. If no recipient for a message m is left, the pair for m in $wire_n$ can be cleaned up by the internal task *discard*. However, as long as the pair for a message m is contained in $wire_n$, it is possible that the set of recipients is altered by the task *misbehave*. Modeling the unreliable behavior of the network, *misbehave* can add processes to or remove processes from the current set of intended recipients for m. This includes adding new recipients to an empty set or removing all recipients. The only constraint at this point is that the task is fair, that it does not have any bias towards particular messages. The likelihood for being altered has to be the same for all pairs in $wire_n$ to ensure that messages sent infinitely often are also delivered infinitely often and that messages sent finitely often are also delivered finitely often. Note that the network as specified here does not invent completely new messages, which includes that it does not corrupt sent messages. This is actually more restrictive than necessary. All messages in Hybster are authenticated (see below), which ensures their integrity at the protocol level even if the network could deliver corrupted messages.

3.4 Cryptography

Hybster relies on digital signatures and message authentication codes (MACs) to authenticate messages exchanged over the network. Moreover, cryptographic hash functions are used to reduce the amount of transmitted data.

The authenticity of messages is documented by message certificates (Listing 5a). Message certificates are messages themselves, they are tuples comprising a certificate type, a proof of their validity, and further elements if required. Usually, they are directly attached to the mes-

Listing 4: Specification of the communication system shared by all processes.

(a)

```
1 module Message
2   extends Tuple⟨MessageType, ...⟩

4   declare output method type() → MessageType
```

(b)

```
1 module MessageSink
2   declare input call receive(m ∈ Message)
```

(c)

```
1 module Network_n

3 upon init() do
4   nodes_n ∈ P(MessageSink) ≔ set of all connected processes
5   wire_n ∈ P(Message × P(MessageSink)) ≔ {}

7 upon input call send(m ∈ Message, R ∈ P(MessageSink))
8   asserts
9     R ⊆ nodes_n
10  do
11    if ⟨m, R*⟩ ∈ wire_n then
12      wire_n ≔ wire_n \ {⟨m, R*⟩} ∪ {⟨m, R* ∪ R⟩}
13    else
14      wire_n ≔ wire_n ∪ {⟨m, R⟩}

16 upon internal task transmit(m ∈ Message, r ∈ MessageSink)
17  with
18    ∃⟨m, R⟩ ∈ wire_n : r ∈ R
19  do
20    wire_n ≔ wire_n \ {⟨m, R⟩} ∪ {⟨m, R \ {r}⟩}
21    output r.receive(m)

23 upon internal task discard(m ∈ Message)
24  with
25    ∃⟨m, ∅⟩ ∈ wire_n
26  do
27    wire_n ≔ wire_n \ {⟨m, ∅⟩}

29 upon internal task misbehave(m ∈ Message, R ∈ P(MessageSink), R′ ∈ P(MessageSink))
30  with
31    ∃⟨m, R⟩ ∈ wire_n
32    R′ ⊆ nodes_n
33  do
34    wire_n ≔ wire_n \ {⟨m, R⟩} ∪ {⟨m, R′⟩}
```

sage they certify forming a certified message (Listing 5b). The notation for such a message is $\langle \ldots \rangle[certificate]$, with $[certificate]$ being a placeholder for an identifier of the employed certification method. $\langle \ldots \rangle_{\sigma_x}$ denotes a message signed by a process x. If a message is certified by a MAC, it is written as $\langle \ldots \rangle_{\mu_{x,y,z}}$, where the underlying secret key is shared among the processes x, y, and z. If a process x issues an array of MACs, a so-called authenticator [7] for the processes y and z, the message is annotated with $\langle \ldots \rangle_{\alpha_{x:y,z}}$.

Listing 5: Specification of certified messages.

(a)

```
1 module MessageCertificate
2   extends Message

4   declare output method proof() → Proof
```

(b)

```
1 module CertifiedMessage
2   extends Message

4   declare output method certificate() → MessageCertificate
```

Listings 6a and 6b depict the specifications for providers of signatures and MACs, respectively. An instance of the type *SignatureProvider* is initialized with a public/private key pair. It issues digital signatures on the basis of this pair in the method *createSignature* and verifies signatures in $verifySignatures$ using the public key connected to the signature in question. Message authentication codes are issued by $MacProvider$ instances. MACs are created and verified in the methods $createMac$ and $verifyMac$ using the shared key that is configured during the initialization of the instance.

The implementation of a cryptographic hash function is provided by the type *DigestProvider* as shown in Listing 6c. This type offers a global method *digest* that returns the hash calculated for a given argument in form of a message similar to the certificates issued by certification providers described above. The hash function is required to be collision resistant such that $\forall m, m' \colon digest(m) = digest(m') \leftrightarrow m = m'$ holds with almost absolute certainty during the period in which the message m is relevant and not outdated from perspective of the protocol.

3.5 Faults

Constraints. A process, be it a replica or a client, can fail at any point in time. The behavior of a failed process is undetermined, it can stop the processing of messages completely or it can start to act maliciously by sending messages that are not correct with regard to the protocol specification. Thus, processes can fail arbitrarily, that is, they can be Byzantine. However, this does not apply to all components of replicas. Part of each replica is a trusted subsystem that is assumed to fail only by crashing. That is, independent of the rest of the process, it is assumed that a trusted subsystem either behaves according to the specification or does not carry out any operation, it is assumed that a trusted subsystem never yields a result that is incorrect. Nonetheless, a process is still regarded as either correct or faulty. If some of the components of a process are faulty in any way, the whole process is deemed as faulty.

Furthermore, it is assumed that the cryptographic assumptions hold under all circumstances. Even if an adversary controlled all faulty processes in the system, it is assumed that the collaborating processes would not be able to impersonate correct processes or find collisions for message hashes. Therefore, adversaries are regarded as computationally restricted.

Given a point t in the time $\mathcal{T}$ of some execution of the system, $\mathcal{R}_f(t)$ denotes the set of all replicas that are faulty at t and $\mathcal{C}_f(t)$ the set of all faulty clients. The recovery of processes is not considered. Therefore, it holds: $\forall t, t' \in \mathcal{T} \colon t \leq t' \rightarrow (\mathcal{R}_f(t) \cup \mathcal{C}_f(t)) \subseteq (\mathcal{R}_f(t') \cup \mathcal{C}_f(t'))$. Further, if a replica or a client is said to be faulty without a reference to time, there is some

Listing 6: Specification of providers for message certificates and cryptographic hashes.

(a)

```
1  module SignatureProvider_s

3  upon init(pubkey ∈ PublicKey, privkey ∈ PrivateKey) do
4    pubkey_s := pubkey
5    privkey_s := privkey

7  upon output call createSignature(m) → MessageCertificate do
8    p := signature for m using privkey_s
9    return ⟨SIGNATURE, pubkey_s, p⟩

11 upon output call verifySignature(⟨SIGNATURE, pubkey, p⟩, m) → Bool do
12   return use pubkey to verify that p is a valid signature for m
```

(b)

```
1  module MacProvider_s

3  upon init(key ∈ SharedKey) do
4    key_s := key

6  upon output call createMac(m) → MessageCertificate do
7    p := calcMac(m)
8    return ⟨MAC, p⟩

10 upon output call verifyMac(⟨MAC, p⟩, m) → Bool do
11   return p = calcMac(m)

13 upon internal call calcMac(m) → Mac do
14   return MAC for m using key_s
```

(c)

```
1  module DigestProvider

3  upon global call digest(m) → Message
4    d := calculate digest of m
5    return ⟨DIGEST, d⟩
```

time in the execution in question in which it does not behave according to the specification. As a consequence, the set of faulty replicas is given by $\mathcal{R}_f \stackrel{\text{def}}{=} \bigcup_{t \in \mathcal{T}} \mathcal{R}_f(t)$ and the set of faulty clients by $\mathcal{C}_f \stackrel{\text{def}}{=} \bigcup_{t \in \mathcal{T}} \mathcal{C}_f(t)$. It is assumed that the number of faulty replicas does not exceed a particular f, thus it is assumed that $|\mathcal{R}_f| \leq f < n$ is always true. Opposed to that, there is no required upper bound for the number of faulty clients, that is, $|\mathcal{C}_f| \leq nclients$. Complementing the sets of faulty processes, the sets for correct replicas and clients are defined as $\mathcal{R}_c \stackrel{\text{def}}{=} \mathcal{R} \setminus \mathcal{R}_f$ and $\mathcal{C}_c \stackrel{\text{def}}{=} \mathcal{C} \setminus \mathcal{C}_f$, respectively.

Quorums. Since processes can fail silently or omit to send messages and since the assumed upper bounds for message delivery and processing times are not known, a correct process cannot know if an expected message from another process p has not arrived yet because p is faulty or because the network or p are slower than currently assumed. Hence, in situations where messages

are not received in time, processes cannot determine if another process p is faulty or not. Even if p provides a correct message in time, it can be Byzantine or fail at a later point in time. As a consequence, state-machine replication protocols do not rely on single replicas to acknowledge protocol information or states but on sufficiently large subsets of the replica group, so-called *quorums* [7]. Formally, a quorum is defined as a set Q such that $Q \subseteq \mathcal{R} \wedge |Q| \geq qs$, with qs being a minimum quorum size that depends on the properties the particular quorum shall provide. The most important quorums are called *intersecting quorums*. If not stated otherwise, these quorums have the fixed minimum size q and fulfill the properties that (1) there is at least one, possibly faulty, replica in the intersection of every two quorums ($2q > n$) and (2) the number of correct replicas in the system suffices to form a quorum ($n \geq q + f$). An important implication of these properties is that each intersecting quorum contains at least one correct replica ($q > f$). Given a number of replica faults f that has to be tolerated, these properties lead to a minimum configuration of $n = 2f + 1$ required replicas with a minimum size for intersecting quorums of $q = f + 1$. Another type of quorums are *acknowledging quorums*, also called weak quorums. These quorums have the property that at least one correct replica is contained. Thus, their minimum size is always $f + 1$, independent of the number of replicas n. Further, in the minimum configuration they have the same size as intersecting quorums.

4 Specification of Hybster

This section describes the properties Hybster provides and presents a formal specification of the protocol. An informal step-by-step description will be part of a future version of this document.

4.1 System Properties

Hybster provides roughly the same properties as PBFT [5], which is a form of linearizability [10] that takes the arbitrary behavior of Byzantine clients into account. From the perspective of each correct client, the replicated system appears to execute all invocations sequentially in an order that respects the real-time relation between the invocations.

The specification of the properties Hybster satisfies is presented in Listing 7 and is based on the one given by Castro [5, p. 30]. It uses the service and invocation handler specifications introduced in Section 3.1.

Initialization. First, a couple of system-wide variables are initialized ($init()_s$). svc_s takes an instance of the service implementation. The set of issued but yet unprocessed invocations is maintained in $pending_s$ and all undelivered responses in $responses_s$. Invocations are triples of a client ($\mathcal{C}$), an invocation number ($^i\mathbb{N}$), and a service command ($\mathcal{O}$), thus $\mathcal{I} \stackrel{\text{def}}{\equiv} \mathcal{C} \times {}^i\mathbb{N} \times \mathcal{O}$. Invocation numbers are individual to each client and correct clients assign these numbers only once and monotonically increasing. Therefore, invocations can be identified by the pair of a client and an invocation number ($\mathcal{C} \times {}^i\mathbb{N}$). A response to an invocation is formed by its identifier and the value returned by its execution ($\mathcal{O}'$), that is, $\mathcal{Y} \stackrel{\text{def}}{\equiv} \mathcal{C} \times {}^i\mathbb{N} \times \mathcal{O}'$. Moreover, the system-wide variable $last\_inv_s$ references a vector that stores for each client the invocation number of the last executed invocation. All entries of this vector are initialized with zero and the first valid invocation number for each client is one.

Listing 7: Specification of linearizability with Byzantine clients.

```
1  module ByzantineLinearizability_s
2    extends InvocationHandler as c

4  upon init()_s do
5    svc_s := Service()
6    pending_s ∈ P(I) := {}
7    responses_s ∈ P(Y) := {}
8    last_invs_s : C → ^iN, for all c ∈ C do last_invs_s(c) := 0

10 upon init()_c do
11   invno_c ∈ ^iN := 0
12   svccmd_c ∈ O ∪ {∅} := ∅
13   isfaulty_c ∈ Bool := False

15 upon input call startInvocation(svccmd ∈ O)_c
16   asserts
17     svccmd_c = ∅
18   do
19     invno_c := invno_c + 1
20     svccmd_c := svccmd
21     registerInvocation(⟨c, invno_c, svccmd⟩)

23 upon internal call registerInvocation(inv' = ⟨c ∈ C, ...⟩)_s do
24   pending_s := pending_s \ {inv ∈ pending_s | inv = ⟨c, ...⟩} ∪ {inv'}

26 upon internal task processInvocation(inv = ⟨c ∈ C, invno ∈ ^iN, svccmd ∈ O⟩)_s
27   with
28     inv ∈ pending_s ∧ invno > last_invs_s(c) ∧ preservesFairness(inv)
29   do
30     pending_s := pending_s \ {inv}
31     svcret := invoke svc_s.invoke(c, invno, svccmd)
32     responses_s := responses_s \ {resp ∈ responses_s | resp = ⟨c, ...⟩} ∪ {⟨c, invno, svcret⟩}
33     last_invs_s(c) := invno

35 declare internal call preservesFairness(inv = ⟨c ∈ C, invno ∈ ^iN, svccmd ∈ O⟩)_s

37 upon output task invocationCompleted(svcret ∈ O')_c
38   with
39     isfaulty_c ∨ resp = ⟨c, invno_c, svcret⟩ ∈ responses_s
40   do
41     responses_s := responses_s \ {resp}
42     svccmd_c := ∅

44 upon internal task clientFailureOccured()_c
45   with
46     ¬isfaulty_c
47   do
48     isfaulty_c := True

50 upon internal task injectFaultyInvocation(c ∈ C, invno ∈ ^iN, svccmd ∈ O)_c
51   with
52     isfaulty_c
53   do
54     registerInvocation(⟨c, invno, svccmd⟩)
```

Invocation. Invocation handler instances are dedicated to single clients and used by them to invoke operations of the replicated service. A handler instance maintains three variables (see $init()_c$), the invocation number of the last invocation $invno_c$, $svccmd_c$ that stores the service command of a currently running but not yet completed invocation or $\varnothing$ if no invocation is in progress, and the flag $isfaulty_c$ that marks faulty clients. Correct clients start only one invocation at a time and wait for its completion before they start a new one. Hence, the input method *startInvocation* expects that $svccmd_c$ is $\varnothing$ when it is called. Subsequently, it assigns the next invocation number to the passed service command, sets $svccmd_c$ to mark the invocation as in progress, and submits the new invocation by calling *registerInvocation*. This system-wide method adds invocations to the set of pending invocations while ensuring that each client has at most one outstanding invocation in that set.

Processing. Pending invocations are delivered to the service instance by the task *process-Invocation*, one invocation at a time. It considers only invocations with numbers higher than the numbers stored in $last\_inv_s$, that is, with numbers higher than the one of the last invocation that was executed for the respective client. Being always the case for correct clients, this prevents faulty clients from injecting invocations in a non-increasing order. Further, *processInvocation* makes use of the method *preservesFairness* to select the invocation to be executed next. This auxiliary method is only declared and documents the requirement that the selection mechanism is unbiased and does not give particular clients an advantage. Once an invocation has been chosen, it is removed from the pending set and handed over to the service implementation. The value returned by the service is used to create a response. Before this response is added to the set of undelivered responses, older responses for the client are removed if necessary. A faulty client could have sent a new invocation without collecting the response for a previous one. Finally, $last\_inv_s$ is updated to mark the current invocation as executed.

Completion. Adhering to the interface specified by the type $InvocationHandler$, invocations are delivered to clients through the output task *invocationCompleted*. Provided that a client c is not marked as faulty, this task removes a response belonging to c from the set of undelivered responses and marks the invocation as completed by setting $svccmd_c$ to $None$ before it delivers the return value.

Faulty Clients. A client c can fail at any point in time. This is modeled by the task *clientFailureOccured*. If executed, it simply sets the flag $isfaulty_c$ to $True$. From that time on, the client c can circumvent the regular *startInvocation* method and can issue arbitrary invocation as realized by the task *injectFaultyInvocation*. However, it can still only do so for invocations that are attributed to c itself. As stated in Section 3.5, the system model assumes that even arbitrarily behaving clients are not able to impersonate correct clients. In addition to the injection of arbitrary invocation, a faulty client can locally deliver any return value as reflected in the task *invocationCompleted*.

4.2 Auxiliary Modules

The specifications of Hybster's main protocols make use of two mechanisms that help to cope with the partially synchronous environment and the unreliable network as assumed by the system model (Section 3): adaptive timeouts and retransmitting connections. The respective modules realizing these mechanisms are presented in the following.

4.2.1 Adaptive Timeouts

The time model Hybster's specification is based on assumes upper bounds for message delivery and processing times, upper bounds, however, that are unknown (cf. Section 3.2). As a consequence, it cannot be known how long certain protocol actions need to finish. Nonetheless, it is known that they do not require an infinite amount of time as long as their completion does not depend on a faulty process. Therefore, provided that some protocol action can be retried infinitely often, it is possible to approach the upper bound by increasing the value for the timeout that determines when the action is regarded as failed.

Listing 8: Specification of timeouts adaptive to unknown upper bounds for delays.

```
1  module AdaptiveTimeout_s

3  upon init(procs ∈ Set) do
4    procs_s := procs
5    timer_s := Timer()
6    delay_delta_s ∈ ℕ := increment if timeout expired
7    delay_s ∈ ℕ := delay_delta_s
8    expired_s ∈ Bool := False

10 upon input call start(x)
11   asserts
12     x ∈ procs_s // x could be used in more sophisticated implementations
13   do
14     adapt()
15     output timer_s.schedule(delay_s)

17 upon input call adapt() do
18   if expired_s then
19     delay_s := delay_s + delay_delta_s
20     expired_s := False

22 upon input call stop() do
23   output timer_s.cancel()
24   expired_s := False

26 upon internal task timerExpired()
27   with
28     input timer_s.timerExpired()
29   do
30     expired_s := True

32 upon output call isExpired() → Bool do
33   return expired_s

35 upon output call isRunning() → Bool do
36   return input timer_s.isScheduled()
```

This mechanism is realized in the module *AdaptiveTimeout* shown in Listing 8. It is intended that one instance of this module is used for one particular protocol action that has to be carried out by one process from a group of processes. For example, if a replica is supposed to provide a particular message, an instance of this module can be used to signal when this step in the protocol is considered as failed. If this is the case, the step is repeated, but this time, a different replica is chosen to provide the message and it is given more time to do so. Eventually, this

step will complete successfully, at the latest when a correct replica is selected and the currently configured timeout exceeded the upper bound for providing the expected message.

Although the implementation of *AdaptiveTimeout* is tailored to the time model stated above, please note that *AdaptiveTimeout* in fact abstracts from the concrete model. Currently, it increases the timeout value linearly. If a time model as in PBFT were used in which a delay function $delay(t)$ does not grow faster than the time t [5], an exponential increase could compensate for that. More sophisticated strategies would be required if the time bounds were only assumed to hold long enough such that the system is able to make progress [9]. Still, higher-level protocols could rely on the abstraction provided by *AdaptiveTimeout*.

The module *AdaptiveTimeout* in detail: Each instance maintains a reference to a timer instance, the current timeout value stored in $delay_s$ and initialized with a configured constant $delay\_delta_s$, and a flag $expired_s$ set when the timeout expired. The instance also gets the set of processes that are within the considered group. This set is actually not required by the presented implementation and is only provided to allow for other strategies that adjust the timeout value for processes individually instead of globally for all processes. The timeout is started through the input method *start*. It takes the process as argument that is currently expected to carry out the monitored action. Again, the presented implementation does not make use of this information. Upon being invoked, the method *start* configures the timer to signal after time $delay_s$. Prior to that, it is checked if the current timeout value $delay_s$ needs to be adjusted. The strategy for that is realized by the input method *adapt*. If the timer was marked as expired, it adds $delay\_delta_s$ to $delay_s$ and resets the expired flag. If the timed protocol action completed or if the timeout is not needed anymore, the input method *stop* can be invoked canceling a currently running timer. Otherwise, the timer will eventually expire, monitored with the internal *timerExpired* task. This sets the expired flag which can be queried by the output method *isExpired*. Additionally, *isRunning* returns if the timer is currently scheduled or not.

4.2.2 Connections

Even if a process expected to provide some message were correct and even if other processes waited actually long enough to receive this message, the message might not come in time since the network could have dropped it. The upper bounds for message delivery only hold if the message is sent sufficiently, virtually infinitely often (see Section 3.3).

Retransmitting messages is the responsibility of the *Connection* module presented in Listing 9. It sends registered messages over and over again until they are explicitly removed from the connection by a higher-level protocol. An instance of the *Connection* module expects a set of recipients when constructed. It further has a reference to the network and keeps with out_s a set of all messages that are currently in transmission. Messages are handed over to the connection via the *send* input method which adds the message to out_s. The internal task *retransmit* is enabled as long as some message is contained in out_s. When executed, it picks one message and (re-)sends it to the configured recipients. Nevertheless, the selected message remains in the set out_s and could be chosen again in a later execution of the task. If multiple messages are contained in out_s, the selection mechanism is supposed to be fair, on average all messages are retransmitted at the same rate. The module offers multiple variants of the method *remove* used to deregister messages from the connection. *remove* can be invoked with a single message, a set of messages, or a predicate that determines which messages are to be removed from out_s.

Listing 9: Specification of a retransmitting connection masking message loss.

```
1  module Connection_s

3  upon init(recipients ∈ P(MessageSink)) do
4    net_s ∈ Network := reference to the network
5    recipients_s := recipients
6    out_s ∈ P(Message) := {}

8  upon input call send(m ∈ Message) do
9    out_s := out_s ∪ {m}

11 upon internal task retransmit(m ∈ Message)
12   with
13     m ∈ out_s
14   do
15     output net_s.send(m, recipients_s)

17 upon input call remove(m ∈ Message) do
18   out_s := out_s \ {m}

20 upon input call remove(M ∈ P(Message)) do
21   out_s := out_s \ M

23 upon input call remove(P: Message → Bool) do
24   out_s := out_s \ {m ∈ out_s | P(m)}
```

4.3 TrInX

Listing 10: Specification of the trusted subsystem TrInX.

```
1  module TrInX_s

3  upon init(id ∈ TssID, ctrtypes ∈ Set, key ∈ SharedKey) do
4    inst_s := id
5    ctrtypes_s := ctrtypes
6    macprov_s := MacProvider(key)
7    ctrs_s : ctrtypes → ℕ, for all tc ∈ ctrtypes do ctrs_s(tc) := 0

9  upon public call createContinuingCertificate(tc, tv' ∈ ℕ, m) → MessageCertificate
10   asserts
11     tc ∈ ctrtypes_s ∧ tv' ≥ ctrs_s(tc)
12   do
13     tv := ctrs_s(tc)
14     p := macprov_s.createMac(inst_s‖tc‖tv'‖tv‖m)
15     ctrs_s(tc) := tv'
16     return ⟨TCTR, inst_s, tc, tv', tv, p⟩

18 upon public call createIndependentCertificate(tc, tv' ∈ ℕ, m) → MessageCertificate
19   asserts
20     tc ∈ ctrtypes_s ∧ tv' > ctrs_s(tc)
21   do
22     p := macprov_s.createMac(inst_s‖tc‖tv'‖ − ‖m)
23     ctrs_s(tc) := tv'
24     return ⟨TCTR, inst_s, tc, tv', −, p⟩

26 upon output call verifyCertificate(⟨TCTR, inst, tc, tv', tv, p⟩, m) → Bool do
27     return macprov_s.verifyMac(inst‖tc‖tv'‖tv‖m)

29 upon input call forwardCounter(tc, tv' ∈ ℕ)
30   asserts
31     tc ∈ ctrtypes_s ∧ tv' ≥ ctrs_s(tc)
32   do
33     ctrs_s(tc) := tv'
```

4.4 Processes

4.4.1 Clients

Listing 11: Specification of invocation handlers dedicated to a client.

```
1  module ByzInvocationHandler_c
2    extends InvocationHandler
3    extends MessageSink

5  upon init(pubkey ∈ PublicKey, privkey ∈ PrivateKey, pubkeys: R → PublicKey) do
6    repconns_c: R → Connection, for all r ∈ R do repconns_c(r) := Connection({r})
7    repgrpconn_c := Connection(R)
8    sigprov_c := SignatureProvider(pubkey, privkey)
9    pubkeys_c := pubkeys
10   im_c := InvocationHandler_c::Invocation()

12 upon input call startInvocation(svccmd ∈ O) do
13   output im_c.startInvocation(svccmd)

15 upon output task invocationCompleted(svcret ∈ O') with
16   input im_c.invocationCompleted(svcret)

18 upon input call receive(m ∈ Message) do
19   output im_c.receive(m)

21 upon internal call hasValidCertificate(m = ⟨...⟩_σr) → Bool do
22   p := m.certificate().proof()
23   return input sigprov_c.verifySignature(⟨SIGNATURE, pubkeys_c(r), p⟩, m)
```

4.4.2 Replicas

Listing 12: Specification of the current view status used by protocol modules.

```
1  module ViewStatus_s

3  upon init(replica ∈ R) do
4    replica_s ∈ R := replica
5    v_stab_s ∈ ᵛN := 0
6    v_cur_s ∈ ᵛN := v_stab_s

8  upon global call leader(v ∈ ᵛN) → R do
9    return r_(v mod n)

11 upon input call leaveViewFor(v' ∈ ᵛN)
12   asserts
13     v_cur_r < v'
14   do
15     v_cur_r := v'

17 upon input call enterView(v' ∈ ᵛN)
18   asserts
19     v_stab_r < v' ≤ v_cur_r
20   do
21     v_stab_r := v'

23 upon output call latestLeader() → R do
24   return leader(v_stab_s)
```

```
26 upon output call isInStableView() → Bool do
27   return v_cur_s = v_stab_s

29 upon output call isStableLeader() → Bool do
30   return isInStableView() ∧ replica_s = lastestLeader()

32 upon output call isStableFollower() → Bool do
33   return isInStableView() ∧ replica_s ≠ lastestLeader()

35 upon output call stableView() → ᵛℕ do
36   return v_stab_s

38 upon output call currentView() → ᵛℕ do
39   return v_cur_s
```

Listing 13: Specification of a replica making use of multiple protocol modules.

```
1 module Replica_r
2   extends MessageSink

4 def TC ≡ def {𝔐, 𝔒, 𝔑}

6 upon init(pubkey ∈ PublicKey, privkey ∈ PrivateKey,
      tsskey ∈ SharedKey, pubkeys: R ∪ C → PublicKey, tssinsts: R → TssID) do
7   repconns_r: R \ {r} → Connection, for all r ∈ R \ {r} do repconns_r(r) := Connection({r})
8   repgrpconn_r := Connection(R \ {r})
9   tss_r := TrInX(tssinsts(r), TC, tsskey)
10  tssinsts_r := tssinsts
11  sigprov_r := SignatureProvider(pubkey, privkey)
12  pubkeys_r := pubkeys
13  cm_r := Replica_r::Client
14  om_r := Replica_r::Ordering
15  km_r := Replica_r::Checkpointing
16  vm_r := Replica_r::ViewChange
17  em_r := Replica_r::Execution
18  sm_r := Replica_r::StateTransfer

20 upon input call receive(m ∈ Message) do
21   // dispatch message according to message type
22   output (cm_r|om_r|km_r|vm_r|em_r|sm_r).receive(m)

24 upon internal call hasValidCertificate(m = ⟨...⟩_σx) → Bool do
25   p := m.certificate().proof()
26   return input sigprov_c.verifySignature(⟨SIGNATURE, pubkeys_r(x), p⟩, m)

28 upon internal call hasValidCertificate(m = ⟨...⟩_τ(r',tc,tv',tv)) → Bool do
29   p := m.certificate().proof()
30   return input tss_r.verifyCertificate(⟨TCTR, tssinsts_r(r'), tc, tv', tv, p⟩, m)
```

4.5 Invocation Protocol

4.5.1 Client Side

Listing 14: Specification of the client role of the invocation protocol.

```
1  module InvocationHandler_c::Invocation

3  upon internal call leader(v ∈ ᵛℕ) → ℛ do
4    return r_{v mod n}

6  upon init() do
7    v_c ∈ ᵛℕ := 0
8    invno_c ∈ ⁱℕ := 0
9    req_c ∈ Message ∪ {∅} := ∅
10   reps_c ∈ 𝒫(Message) := {}
11   rq_timeout_c := AdaptiveTimeout(ℛ)
12   retrans_c ∈ Bool := False

14 upon input call startInvocation(svccmd ∈ 𝒪)
15   asserts
16     req_c = ∅
17   do
18     invno_c := invno_c + 1
19     req_c := ⟨REQUEST, c, invno_c, svccmd⟩_{σ_c}
20     sigprov_c.createSignature(req_c)
21     output repconns_c(leader(v_c)).send(req_c)
22     rq_timeout_c.start(leader(v_c))

24 upon internal task retransmit()
25   with
26     req_c ≠ ∅ ∧ ¬retrans_c ∧ rq_timeout_c.isExpired()
27   do
28     rq_timeout_c.adapt()
29     output repgrpconn_c.send(req_c)
30     retrans_c := True

32 upon global call isCorrectReply(rp, r ∈ ℛ, v ∈ ᵛℕ, c' ∈ 𝒞, invno ∈ ⁱℕ, svcret ∈ 𝒪') do
33   return rp = ⟨REPLY, r, v, c', invno, svcret⟩_{σ_r} ∧ hasValidCertificate(rp)

35 upon input call receive(rp' = ⟨REPLY, r, v', ·, ·, svcret'⟩_...) do
36   if req_c ≠ ∅ ∧ isCorrectReply(rp', ·, ·, c, invno_c, ·) then
37     if ∄v ∄svcret: rp = ⟨REPLY, r, v, c, invno_c, svcret⟩_... ∈ reps_c then
38       reps_c := reps_c ∪ {rp'}
39     else if svcret' ≠ svcret then
40       // r is faulty
41     else if v' > v then
42       reps_c := reps_c ∪ {rp'} \ {rp}

44 upon global call isCorrectReplyCertificate(Y ⊆ Message, c' ∈ 𝒞, invno ∈ ⁱℕ, svcret ∈ 𝒪' do
45   return |Y| > f
46            ∧ (∀rp ∈ Y ∃r ∈ ℛ: isCorrectReply(rp, r, ·, c', invno, svcret)
47              ∧ (∀rp' ∈ Y: rp' = ⟨REPLY, r, ...⟩_... → rp' = rp))

49 upon output task invocationCompleted(svcret ∈ 𝒪')
50   with
51     ∃Y ⊆ reps_c: isCorrectReplyCertificate(Y, c, invno_c, svcret)
52   do
```

```
53      rq_timeout_c.stop()
54      output repconns_c(leader(v_c)).remove(req_c)
55      if retrans_c then
56        output repgrpconn_c.remove(req_c)
57      v_c ≔ max({v | ⟨Reply, ·, v, ...⟩... ∈ Y})
58      req_c ≔ ∅
59      reps_c ≔ {}
60      retrans_c ≔ False
```

4.5.2 Replica Side

Listing 15: Specification of the replica role of the invocation protocol.

```
1  module Replica_r::Client

3  def global ReturnEntry ≡^def ^i ℕ × (O' ∪ {∅})

5  upon global call invocationNumber(⟨invno, svcret⟩ ∈ ReturnEntry) → ^i ℕ do
6    return invno

8  upon init() do
9    cstatus_r ≔ ViewStatus(r)
10   reqs_r ∈ P(Message) ≔ {}
11   rq_timeouts_r : C → AdaptiveTimeout
12   last_retvals_r : C → ReturnEntry
13   cliconns_r : C → Connection
14   for all c ∈ C do
15     rq_timeouts_r(c) ≔ AdaptiveTimeout(R)
16     last_retvals_r(c) ≔ ⟨0, ∅⟩
17     cliconns_r(c) ≔ Connection({c})
18   contact_r ≔ cstatus_r.latestLeader()

20 upon global call isCorrectRequest(rq, c ∈ C, invno ∈ ^i ℕ, svccmd ∈ O) do
21   return rq = ⟨Request, c, invno, svccmd⟩_σc ∧ hasValidCertificate(rq)

23 upon input call receive(rq' = ⟨Request, c, invno', svccmd⟩...) do
24   if isRelevantRequest(rq') ∧ isCorrectRequest(rq', ...) then
25     storeRequest(rq')
26     if cstatus_r.isStableFollower() then
27       forwardRequest(rq')
28     startRequestTimeout(c)

30 upon internal call isRelevantRequest(rq' = ⟨Request, c, invno', ·⟩...) do
31   return ¬isRequestExecuted(rq') ∧ (∄rq ∈ reqs_r ∃invno ≥ invno' : rq = ⟨Request, c, invno, ·⟩...)

33 upon internal call storeRequest(rq' = ⟨Request, c, ...⟩...) do
34   reqs_r ≔ reqs_r \ {rq ∈ reqs_r | rq = ⟨Request, c, ...⟩...}
35   reqs_r ≔ reqs_r ∪ {rq'}

37 upon internal call forwardRequest(rq = ⟨Request, c, ·⟩...) do
38   output repconns_r(contact_r).remove({m | m = ⟨Request, c, ...⟩...})
39   output repconns_r(contact_r).send(rq)

41 upon output call pendingRequests() → P(Message) do
42   return reqs_r

44 upon input call requestExecuted(rq' = ⟨Request, c, invno', svccmd⟩..., svcret ∈ O') do
```

```
45  reqs_r ≔ reqs_r \ {rq | rq = ⟨REQUEST, c, invno, ·⟩... ∧ invno ≤ invno'}
46  last_retvals_r(c) ≔ ⟨invno', svcret⟩
47  sendReply(c, invno', svcret)
48  if ∄⟨REQUEST, c, ...⟩... ∈ reqs_r ∧ rq_timeouts_r(c).isRunning() then
49    rq_timeouts_r(c).stop()

51 upon output call isRequestExecuted(rq = ⟨REQUEST, c, invno, ·⟩...) do
52  return invno ≤ invocationNumber(last_retvals_r(c))

54 upon internal call sendReply(c ∈ C, invno ∈ ⁱℕ, svcret ∈ O') do
55  rp ≔ ⟨REPLY, r, cstatus_r.stableView(), c, invno, svcret⟩_σr
56  sigprov_r.createSignature(rp)
57  output cliconns_r(c).remove({m | m = ⟨REPLY, r, ·, c, ...⟩...})
58  output cliconns_r(c).send(rp)

60 upon output call createReturnValueMapSnapshot() → (C → ReturnEntry) do
61  return last_retvals_r

63 upon input call installReturnValueMap(retvals ∈ C → ReturnEntry) do
64  last_retvals_r ≔ retvals
65  for all rq = ⟨REQUEST, c, ...⟩... ∈ reqs_r do
66    if isRequestExecuted(rq) then
67      reqs_r ≔ reqs_r \ {rq}
68    if rq_timeouts_r(c).isRunning() then
69      rq_timeouts_r(c).stop()

71 upon internal call startRequestTimeout(c ∈ C) do
72  if rq_timeouts_r(c).isRunning() then
73    rq_timeouts_r(c).stop()
74  rq_timeouts_r(c).start(contact_r)

76 upon output call suspectsLeader() do
77  return ∃to ∈ rq_timeouts_r : to.isExpired()

79 upon input call leaveViewFor(v' ∈ ᵛℕ) do
80  cstatus.leaveViewFor(v')

82 upon input call enterView(v' ∈ ᵛℕ) do
83  cstatus.enterView(v')
84  if cstatus.isInStableView then
85    output repconns_r(contact_r).remove({rq | rq = ⟨REQUEST, ...⟩...})
86    contact_r ≔ cstatus.latestLeader()
87    for all rq = ⟨REQUEST, c, ...⟩... ∈ reqs_r do
88      if cstatus.isStableFollower() then
89        forwardRequest(rq)
90      startRequestTimeout(c)
```

4.6 Ordering Protocol

Listing 16: Specification of the ordering protocol.

```
1  module Replica_r::Ordering

3  uses vm_r ∈ Replica_r::ViewChange
4  uses cm_r ∈ Replica_r::Client
5  uses em_r ∈ Replica_r::Execution

7  // An abort state comprises a prepared set P and
8  // the value of the trusted counter 𝔒 at the time of the abort.
9  def global AbortState ≡^def (P ∈ 𝒫(Message)) × ℕ

11 let order_wnd ∈ ℕ_1

13 upon global call orderWindowSize() do
14   return order_wnd

16 upon init() do
17   ostatus_r := ViewStatus(r)
18   omsgs_r ∈ 𝒫(Message) = {}
19   o_act_r ∈ ^o ℕ := 0
20   o_base_r ∈ ^o ℕ := 0
21   o_comm_r ∈ ^o ℕ := o_base_r
22   o_prep_r ∈ ^o ℕ := o_comm_r
23   o_max_r ≡^def o_base_r + order_wnd

25 upon internal task proposalReady(rq = ⟨REQUEST, c, invno, svccmd⟩_σc)
26   with
27     o' = o_prep_r + 1
28     ostatus_r.isStableLeader() ∧ o' ≤ o_max_r
29     v = ostatus_r.stableView()
30     rq ∈ input cm_r.pendingRequests() ∧ (∄o: ⟨PREPARE, r, v, o, rq⟩_... ∈ omsgs_r)
31   do
32     sendPrepare(v, o', rq)
33     o_prep_r := o'
34     o_act_r := o'

36 upon internal call sendPrepare(v ∈ ^v ℕ, o ∈ ^o ℕ, rq ∈ Message) do
37   pr := ⟨PREPARE, r, v, o, rq⟩_τ(r,𝔒,v|o,−)
38   invoke tss_r.createIndependentCertificate(𝔒, v|o, pr)
39   omsgs_r := omsgs_r ∪ {pr}
40   output repgrpconn_r.send(pr)

42 upon global call isCorrectPrepare(pr, l ∈ ℛ, v ∈ ^v ℕ, o ∈ ^o ℕ, rq ∈ Message) do
43   return l = leader(v) ∧ pr = ⟨PREPARE, l, v, o, rq⟩_τ(l,𝔒,v|o,−) ∧ hasValidCertificate(pr)
          ∧ isCorrectRequest(rq, ...)

45 upon input call receive(pr = ⟨PREPARE, l, v, o, rq⟩_...) do
46   if l ≠ r ∧ v = ostatus_r.stableView() ∧ inOrderWindow(o)
          ∧ isCorrectPrepare(pr, ...) then
47     omsgs_r := omsgs_r ∪ {pr}

49 upon internal task nextPrepared(pr = ⟨PREPARE, l, v, o', rq⟩_...)
50   with
51     ostatus_r.isStableFollower()
```

```
52      pr ∈ omsgs_r ∧ o' = o_prep_r + 1
53    do
54      o_prep_r := o'
55      sendCommit(v, o', digest(rq))
56      o_act_r := o'

58  upon internal call sendCommit(v ∈ ᵛℕ, o ∈ ᵒℕ, d ∈ ⟨DIGEST ...⟩) do
59    co := ⟨COMMIT, r, v, o, d⟩_τ(r,𝔒,v|o,−)
60    invoke tss_r.createIndependentCertficate(𝔒, v|o, co)
61    omsgs_r := omsgs_r ∪ {co}
62    output repgrpconn_r.send(co)

64  upon global call isCorrectCommit(co, r' ∈ ℛ, v ∈ ᵛℕ, o ∈ ᵒℕ, d ∈ ⟨DIGEST ...⟩) do
65    return r' ≠ leader(v) ∧ co = ⟨COMMIT, r', v, o, d⟩_τ(r',𝔒,v|o,−) ∧ hasValidCertificate(co)

67  upon input call receive(co = ⟨COMMIT, r', v, o, d⟩_...) do
68    if r' ≠ r ∧ v = ostatus_r.stableView() ∧ ¬isCommitted(o) ∧ inOrderWindow(o)
          ∧ isCorrectCommit(co, ...) then
69      omsgs_r := omsgs_r ∪ {co}

71  upon global call isCorrectCommittedCertificate(O ⊆ Message, v ∈ ᵛℕ, o ∈ ᵒℕ, rq ∈ Message) do
72    return |O| ≥ q ∧ (∃pr ∈ O: isCorrectPrepare(pr, ·, v, o, rq)
          ∧ (∀m ∈ O: m = pr ∨ isCorrectCommit(m, ·, v, o, digest(rq))))

74  upon internal task nextCommitted(rq = ⟨REQUEST, c, t, svccmd⟩_...)
75    with
76      o' = o_comm_r + 1
77      ∃O ⊆ omsgs_r: isCorrectCommittedCertificate(O, ·, o', rq)
78    do
79      markCommitted(o')
80      output em_r.executeRequest(o', rq)

82  upon input call forwardOrderWindow(o' ∈ ᵒℕ)
83    asserts
84      o' > o_base_r
85    do
86      o_base_r := o'
87      discardOrderMessages()
88      if ¬isCommitted(o') then
89        markCommitted(o')

91  upon internal call discardOrderMessages() do
92    omsgs_r := {m ∈ omsgs_r | inOrderWindow(orderNumber(m))}
93    output repgrpconn_r.remove({m | (m = ⟨PREPARE, ...⟩_... ∨ m = ⟨COMMIT, ...⟩_...) ∧ m ∉ omsgs_r)})

95  upon internal call markCommitted(o ∈ ᵒℕ)
96    asserts
97      o > o_comm_r
98    do
99      if o > o_prep_r then
100       o_prep_r := o
101     o_comm_r := o

103 upon output call isCommitted(o ∈ ᵒℕ) do
104   return o ≤ o_comm_r

106 upon output call lastCommitted() do
```

107 **return** o_comm_r

109 **upon output call** $inOrderWindow(o \in {}^{o}\mathbb{N})$ **do**
110 **return** $o\_base_r < o \leq o\_max_r$

112 **upon output call** $orderWindowBase()$
113 **return** o_base_r

115 **upon output call** $obtainOrderingState(o\_base \in {}^{o}\mathbb{N})$ **do**
116 **return** $\{m \in omsgs_r \mid orderNumber(m) > o\_base\}$

118 **upon global call** $isCorrectOrderingState(I \subseteq Message, v \in {}^{v}\mathbb{N}, o\_base \in {}^{o}\mathbb{N})$ **do**
119 **return** $\forall m \in I\ \exists o\colon (isCorrectPrepare(m, \cdot, v, o, \cdot) \vee isCorrectCommit(m, \cdot, v, o, \cdot))$
$\wedge\ o\_base < o \leq o\_base + order\_wnd$

121 **upon input call** $installOrderingState(I \subseteq Message)$
122 **asserts**
123 $isCorrectOrderingState(I, \cdot, ostatus_r.stableView())$
124 **do**
125 $omsgs_r \coloneqq omsgs_r \cup \{m \in I \mid inOrderWindow(orderNumber(m))\}$

127 **upon output call** $createAbortState() \to AbortState$ **do**
128 $P \coloneqq \{\langle \textsc{Prepare}, \cdot, \cdot, o, \cdot\rangle_{\dots} \in omsgs_r \mid o - 1 = o\_base_r \vee \langle \textsc{Prepare}, \cdot, \cdot, o-1, \cdot\rangle_{\dots} \in omsgs_r\}$
129 **return** $\langle P, ostatus_r.currentView() \mid o\_act_r\rangle$

131 **upon input call** $abortView(v' \in {}^{v}\mathbb{N}, m \in \langle \dots \rangle_{\tau(\dots)})$ **do**
132 **invoke** $tss_r.createConsecutiveCertficate(\mathfrak{O}, v'|0, m)$
133 $o\_act_r \coloneqq 0$
134 $ostatus_r.leaveViewFor(v')$

136 **upon global call** $isCorrectAbortState($
$b = \langle P, tv\_last\rangle \in AbortState, v \in {}^{v}\mathbb{N}, v' \in {}^{v}\mathbb{N}, o\_base \in {}^{o}\mathbb{N}, m \in \langle \dots \rangle_{\tau(r', \mathfrak{O}, v'|0, tv\_last)})$ **do**
137 **return** $hasValidCertificate(m) \wedge isCorrectAbortState(b, v, v', o\_base)$

139 **upon global call** $isCorrectAbortState(\langle P, tv\_last\rangle \in AbortState, v \in {}^{v}\mathbb{N}, v' \in {}^{v}\mathbb{N}, o\_base \in {}^{o}\mathbb{N})$ **do**
140 **return** $tv\_last = \langle v, o\_ctr\rangle$
141 $\wedge\ (\exists o\_max\colon isCorrectPrepareSet(P, v, o\_base, o\_max) \wedge o\_max \geq o\_ctr)$

143 **upon global call** $isCorrectPrepareSet(P, v \in {}^{v}\mathbb{N}, o\_base \in {}^{o}\mathbb{N}, o\_max \in {}^{o}\mathbb{N})$ **do**
144 **return** $|P| = 0 \wedge o\_max = o\_base \vee 0 < |P| \leq order\_wnd$
145 $\wedge\ o\_base = minOrderNumber(P) - 1 \wedge o\_max = maxOrderNumber(P)$
146 $\wedge\ (\forall pr \in P\ \exists o\colon isCorrectPrepare(pr, v, o, \cdot))$
147 $\wedge\ (\forall o' \in {}^{o}\mathbb{N}\colon o\_base < o' \leq o\_max \to \langle \textsc{Prepare}, \cdot, \cdot, o'\cdot\rangle_{\dots} \in P)$

149 **upon input call** $abortView(v' \in {}^{v}\mathbb{N})$ **do**
150 **invoke** $tss_r.forwardCounter(\mathfrak{O}, v'|0)$
151 $o\_act_r \coloneqq 0$
152 $ostatus_r.leaveViewFor(v')$

154 **upon input call** $installAbortState(b = \langle P, tv\_last\rangle \in AbortState)$
155 **asserts**
156 $isCorrectAbortState(b, ostatus_r.stableView(), \dots)$
157 **do**
158 $installOrderingState(P)$

160 **upon output call** $createViewTransition(v' \in {}^{v}\mathbb{N}, B \subseteq AbortState)$ **do**
161 $P^* \coloneqq \{pr \mid \langle P, \cdot\rangle \in B \wedge pr \in P\}$

```
162   if |P*| = 0 then
163     return ∅
164   else
165     o_max ≔ maxOrderNumber(P*)
166     P′ ≔ {⟨Prepare, r, v′, o, m⟩_{τ(r,𝔒,v|o,−)} | o > o_base_r ∧ ⟨Prepare, ·, ·, o, m⟩_... ∈ P*}
167     for o : o_base_r < o ≤ o_max do
168       pr′ ≔ ⟨Prepare, r, v′, o, m⟩_... | pr′ ∈ P′
169       invoke tss_r.createIndependentCertficate(𝔒, v′|o, pr′)
170     return P′

172 upon global call isCorrectViewTransition(P′, v′ ∈ ᵛℕ, o_base ∈ ᵒℕ, B ⊆ AbortState) do
173   P* ≔ {pr | ⟨P, ·⟩ ∈ B ∧ pr ∈ P}
174   return isCorrectPrepareSet(P′, v′, o_base, ·)
175       ∧ (∀pr ∈ P* : pr = ⟨Prepare, ·, ·, o, m⟩_...∧ (o > o_base → ⟨Prepare, ·, v, o, m⟩_... ∈ P′))
176       ∧ (∀pr′ ∈ P′ : pr′ = ⟨Prepare, ·, ·, o, m⟩_...∧ ⟨Prepare, ·, ·, o, m⟩_... ∈ P*)

178 upon input call enterView(v′ ∈ ᵛℕ, P′ ⊆ Message) do
179   ostatus_r.enterView(v′)
180   if |P′| = 0 ∨ r ≠ ostatus_r.lastestLeader() then
181     o_prep_r ≔ o_base_r
182   else
183     o_max ≔ maxOrderNumber(P)
184     o_prep_r ≔ o_max
185     o_act_r ≔ o_max
186   o_comm_r ≔ o_base_r
187   output repgrpconn_r.remove({m | m = ⟨Prepare, ...⟩_... ∨ m = ⟨Commit, ...⟩_...})
188   installOrderingState(P′)

190 upon internal call orderNumber(m ∈ Message)
191   asserts
192     m = (⟨Prepare, ·, ·, o, ·⟩_... ∨ ⟨Commit, ·, ·, o, ·⟩_...)
193   do
194     return o

196 upon internal call maxOrderNumber(I ⊆ Message)
197   return max({o | m ∈ I ∧ o = orderNumber(m)})

199 upon internal call minOrderNumber(I ⊆ Message)
200   return min({o | m ∈ I ∧ o = orderNumber(m)})
```

4.7 Execution

Listing 17: Specification of the service execution.

```
1  module Replica_r::Execution

3  uses cm_r ∈ Replica_r::Client
4  uses km_r ∈ Replica_r::Checkpointing

6  def global ServiceSnapshot ≡^def S × (C → ReturnEntry)

8  upon init() do
9    svc_r := Service()
10   o_exec_r ∈ °N := 0

12 upon input call executeRequest(o ∈ °N, rq = ⟨REQUEST, c, t, svccmd⟩_...)
13   asserts
14     o = o_exec_r + 1
15   do
16     if ¬cm_r.isRequestExecuted(rq) then
17       svcret := invoke svc_r.invoke(c, svccmd)
18       output cm_r.requestExecuted(rq, svcret)
19     o_exec_r := o
20     output km_r.stateReached(o)

22 upon output call createSnapshot() do
23   s := input svc_r.createStateSnapshot()
24   retvals := input cm_r.createReturnValueMapSnapshot()
25   return ⟨s, retvals⟩

27 upon input call installSnapshot(o_exec ∈ °N, ⟨s, retvals⟩ ∈ ServiceSnapshot) do
28   invoke svc_r.installStateSnapshot(s)
29   invoke cm_r.installReturnValueMap(retvals)
30   o_exec_r := o_exec
```

4.8 Checkpointing Protocol

Listing 18: Specification of the checkpointing protocol.

```
1  module Replica_r::Checkpointing

3  uses om_r ∈ Replica_r::Ordering
4  uses em_r ∈ Replica_r::Execution

6  def global
7    CheckpointState ≡^def ^oℕ × ServiceSnapshot
8    CheckpointSet ≡^def {X | ∃o ∀ck ∈ X : ck = ⟨CHECKPOINT, ·, o′, ·⟩... ∧ o = o′}
9    ProvenCheckpoint ≡^def CheckpointSet × ServiceSnapshot

11 let
12   chkpt_int ∈ ℕ_1, chkpt_int ≤ orderWindowSize()
13   retent_wnd ∈ ℕ, retent_wnd < orderWindowSize()

15 upon global call checkpointBase(o)
16   return if o < retent_wnd then 0 else o − retent_wnd

18 upon global call checkpointNumber(⟨o, s*⟩ ∈ CheckpointState) do
19   return o

21 upon global call checkpointState(⟨o, s*⟩ ∈ CheckpointState) do
22   return s*

24 upon global call checkpointNumber(K ∈ CheckpointSet)
25   asserts
26     ∃o ∀⟨CHECKPOINT, ·, o′, ·⟩... ∈ K : o = o′
27   do
28     return o

30 upon init() do
31   kmsgs_r ⊆ Message := {}
32   chkpts_r ⊆ CheckpointState := {}
33   o_stab_r ∈ ^oℕ := 0

35 upon input call stateReached(o ∈ ^oℕ) do
36   if (o mod chkpt_int) = 0 then
37     createCheckpoint(o)

39 upon internal call createCheckpoint(o ∈ ^oℕ) do
40   ks := ⟨o, em_r.createSnapshot()⟩
41   chkpts_r := chkpts_r ∪ {ks}
42   sendCheckpoint(ks)

44 upon internal call sendCheckpoint(⟨o, s*⟩ ∈ CheckpointState)
45   ck := ⟨CHECKPOINT, r, o, digest(s*)⟩_τ(r,𝔐,0,0)
46   invoke tss_r.createConsecutiveCertificate(𝔐, 0, ck)
47   kmsgs_r := kmsgs_r ∪ {ck}
48   output repgrpconn_r.send(ck)

50 upon global call isCorrectCheckpoint(ck, r′ ∈ ℛ, o ∈ ^oℕ, d ∈ ⟨DIGEST ...⟩) do
51   return ck = ⟨CHECKPOINT, r′, o, d⟩_τ(r′,𝔐,0,0) ∧ hasValidCertificate(ck)

53 upon input call receive(ck′ = ⟨CHECKPOINT, r′, o, d⟩...) do
```

```
54   if isRelevantCheckpoint(ck') ∧ isCorrectCheckpoint(ck', ...)
         ∧ (∄d' : ⟨CHECKPOINT, r', o, d'⟩... ∈ kmsgs_r ∧ d' ≠ d) then
55     storeCheckpoint(ck')

57 upon internal call isRelevantCheckpoint(⟨CHECKPOINT, r', o, ·⟩...) do
58   return r' ≠ r ∧ isRelevantCheckpoint(o)

60 upon internal call isRelevantCheckpoint(o ∈ °ℕ) do
61   return o = input om_r.orderWindowBase() ∨ input om_r.inOrderWindow(o)

63 upon internal call storeCheckpoint(ck' = ⟨CHECKPOINT, r', o, d⟩...) do
64   kmsgs_r := kmsgs_r \ {ck ∈ kmsgs_r | ck = ⟨CHECKPOINT, r', o, ·⟩...}
65   kmsgs_r := kmsgs_r ∪ {ck'}

67 upon internal call discardCheckpoints() do
68   o_base := input om_r.orderWindowBase()
69   kmsgs_r := kmsgs_r \ {⟨CHECKPOINT, ·, o, ·⟩... ∈ kmsgs_r | o < o_base}
70   output repgrpconn_r.remove({ck | ck = ⟨CHECKPOINT, ...⟩... ∧ ck ∉ kmsgs_r})

72 upon global call isCorrectCheckpointCertificate(K ⊆ Message, o ∈ °ℕ, d ∈ ⟨DIGEST ...⟩) do
73   return o = 0 ∧ |K| = 0 ∨ |K| > f ∧ (∀ck ∈ K : isCorrectCheckpoint(ck, ·, o, d)

75 upon internal task checkpointStable(o' ∈ °ℕ, d ∈ ⟨DIGEST ...⟩)
76   with
77     o' > o_stab_r ∧ (∃s* : digest(s*) = d ∧ ⟨o', s*⟩ ∈ chkpts_r)
78     ∃K' ⊆ kmsgs_r : isCorrectCheckpointCertificate(K', o', d)
79         ∧ (∀K ⊆ kmsgs_r ∃o : isCorrectCheckpointCertificate(K, o, ·) → o ≤ o')
80   do
81     forwardWindow(o')

83 upon internal call forwardWindow(o' ∈ °ℕ) do
84   o_stab_r := o'
85   chkpts_r := chkpts_r \ {⟨o, ·⟩ ∈ chkpts_r | o < o'}
86   if checkpointBase(o') > input om_r.orderWindowBase() then
87     invoke om_r.forwardOrderWindow(checkpointBase(o'))
88     discardCheckpoints()

90 upon global call isCorrectProvenCheckpoint(pk, o ∈ °ℕ, s* ∈ ServiceSnapshot) do
91   return pk = ⟨C, s*⟩ ∈ ProvenCheckpoint ∧ isCorrectCheckpointCertificate(C, o, digest(s*))

93 upon output call stableCheckpointNumber() do
94   return o_stab_r

96 upon internal call hasStableCheckpoint() do
97   return o_stab_r > 0

99 upon internal call stableCheckpoint() → ProvenCheckpoint
100  asserts
101    hasStableCheckpoint()
102  do
103    ⟨o, s*⟩ ∈ chkpts_r | o = o_stab_r
104    K := {⟨CHECKPOINT, ·, o, digest(s*)⟩... ∈ kmsgs_r}
105    return ⟨K, s*⟩

107 upon output call stableCheckpointCertificate() → CheckpointSet do
108  if ¬hasStableCheckpoint() then
109    return ∅
```

```
110   else
111     ⟨K, ·⟩ ≔ stableCheckpoint()
112     return K

114 upon output call obtainCheckpointingState(o_stab ∈ °ℕ, o_comm ∈ °ℕ) do
115   if o_comm < input om_r.orderWindowBase() then
116     return stableCheckpoint()
117   else if o_stab < o_stab_r then
118     return stableCheckpointCertificate()
119   else
120     return ∅

122 upon global call isCorrectCheckpointingState(ks, o_stab ∈ °ℕ) do
123   return ks = ∅
124       ∨ ks ∈ CheckpointSet ∧ (|ks| = 0 ∨ isCorrectCheckpointCertificate(ks, o_stab, ·))
125       ∨ isCorrectProvenCheckpoint(ks, o_stab, ·)

127 upon input call installCheckpointingState(ks)
128   asserts
129     isCorrectCheckpointingState(ks, ...)
130   do
131     if ks ∈ CheckpointSet then
132       installCheckpointCertificate(ks)
133     else if ks = ⟨K, s*⟩ ∧ ¬ input om_r.isCommitted(checkpointNumber(K)) then
134       o ≔ checkpointNumber(K)
135       output em_r.installSnapshot(o, s*)
136       forwardWindow(o)
137       createCheckpoint(o)
138       installCheckpointCertificate(K)

140 upon input call installCheckpointCertificate(K ∈ CheckpointSet)
141   asserts
142     ∃o: isCorrectCheckpointCertificate(K, o, ·)
143   do
144     if isRelevantCheckpoint(o) then
145       for all ck ∈ K do
146         if isRelevantCheckpoint(ck) then
147           storeCheckpoint(ck)
148       if o > o_stab_r then
149         check checkpointStable()
```

4.9 View-Change Protocol

Listing 19: Specification of the view-change protocol.

```
1  module Replica_r::ViewChange

3  uses cm_r ∈ Replica_r::Client
4  uses om_r ∈ Replica_r::Ordering
5  uses km_r ∈ Replica_r::Checkpointing
6  uses sm_r ∈ Replica_r::StateTransfer

8  upon init() do
9    vstatus_r := ViewStatus(r)
10   v_stab_r ≡^def vstatus_r.stableView()
11   v_cur_r ≡^def vstatus_r.currentView()
12   vmsgs_r ⊆ Message := {}
13   nv_timeout_r := AdaptiveTimeout(R \ {r})

15 upon internal call stableViewCertificate() do
16   asserts
17     v_stab_r > 0
18   do
19     return ⟨New-View, ·, v_stab_r, ...⟩_... ∈ vmsgs_r

21 upon internal call suspectsAcceptedLeader() do
22   return cm_r.suspectsLeader()

24 upon internal call isLeaderSuspectedByQuorum() do
25   return ∃V ⊆ vmsgs_r : |V| > f
26     ∧ (∀vc ∈ V ∃v_to : v_to > v_cur_r ∧ isCorrectViewChange(vc, ·, ·, v_to))

28 upon internal task acceptedLeaderSuspected()
29   with
30     vstatus_r.isInStableView()
           ∧ (suspectsAcceptedLeader() ∨ isLeaderSuspectedByQuorum())
31   do
32     leaveViewFor(v_cur_r + 1)

34 upon internal call leaveViewFor(v' ∈ ^v N) do
35   v := v_cur_r
36   l := leader(v)
37   K := input km_r.stableCheckpointCertificate()
38   b := input om_r.createAbortState()
39   vc := ⟨View-Change, r, v_stab_r, v', K, b⟩_...
40   if nv_timeout_r.isRunning() then
41     nv_timeout_r.stop()
42   output om_r.abortView(v', vc)
43   vstatus_r.leaveViewFor(v')
44   storeViewChange(vc)
45   output repgrpconn_r.send(vc)
46   if v = v_stab_r then
47     output sm_r.startStateRequestTask()

49 upon global call isCorrectViewChange(vc, r' ∈ R, v_from ∈ ^v N, v_to ∈ ^v N) do
50   return ∃K ⊆ Message ∃b ∈ AbortState ∃o_base, o_stab ∈ ^o N :
51       vc = ⟨View-Change, r', v_from, v_to, K, b⟩_... ∧ v_from < v_to
52       ∧ isCorrectCheckpointCertificate(K, o_stab, ·)
```

```
53        ∧ o_base = checkpointBase(o_stab)
54        ∧ isCorrectAbortState(b, v_from, v_to, o_base, vc)

56 upon input call receive(vc' = ⟨View-Change, r', v_from, v_to, K, b⟩...) do
57   if isRelevantViewChange(vc') ∧ isCorrectViewChange(vc', ···) then
58     installViewChange(vc')

60 upon internal call isRelevantViewChange(vc' = ⟨View-Change, r', v_from, v_to, ...⟩...) do
61   return r' ≠ r ∧ (v_to > v_cur_r ∨ ¬vstatus_r.isInStableView ∧ v_to = v_cur_r)

63 upon internal call storeViewChange(vc' = ⟨View-Change, ·, ·, v_to', ···⟩...) do
64   vmsgs_r := vmsgs_r ∪ {vc'}
65   discardViewChanges()

67 upon internal call discardViewChanges() do
68   if vstatus_r.isInStableView() then
69     VC^nxt := ∅
70   else
71     VC^nxt := {vc ∈ vmsgs_r | vc = ⟨View-Change, ·, ·, v_cur_r, ···⟩...}
72   VC^max := {vc_max ∈ vmsgs_r | vc_max = ⟨View-Change, r', ·, v_max, ···⟩...
73 ∧ v_max > v_cur_r
74 ∧ (∀vc ∈ vmsgs_r : vc = ⟨View-Change, r', ·, v, ···⟩... → v ≤ v_max)}
75   VS := {⟨v, V*⟩ | V* ⊆ vmsgs_r ∧ v > v_stab_r
76 ∧ isCorrectViewChangeCertificate(V*, v_stab_r, v)
77 ∧ (∀V ⊆ vmsgs_r :
78 isCorrectViewChangeCertificate(V, v_stab_r, v) → |V| ≤ |V*|)}
79   if v_cur_r ≤ v_stab_r + 1 then
80     V^cur := ∅
81   else if ⟨v_cur_r, V*⟩ ∈ VS then
82     V^cur := V*
83   else
84     V^cur := V* | ⟨v_cur_r − 1, V*⟩ ∈ VS
85   if |VS| = 0 then
86     V^max := ∅
87   else
88     V^max := V* | ⟨v_max, V*⟩ ∈ VS ∧ (∀⟨v, ·⟩ ∈ VS : v ≤ v_max)
89   vmsgs_r := vmsgs_r \ {vc ∈ vmsgs_r | vc = ⟨View-Change, ...⟩...
90 ∧ vc ∉ (VC^nxt ∪ VC^max ∪ V^cur ∪ V^max)}
91   output repgrpconn_r.remove({vc | vc = ⟨View-Change, ...⟩...
92 ∧ vc ∉ (VC^nxt ∪ VC^max ∪ V^cur ∪ V^max)})

94 upon input call installViewChange(vc' = ⟨View-Change, ·, v_from, v_to, K, b⟩...) do
95   storeViewChange(vc')
96   abortStateReceived(v_from, K, b)

98 upon internal call abortStateReceived(v ∈ ᵛℕ, K ∈ CheckpointSet, b ∈ AbortState) do
99   output km_r.installCheckpointCertificate(K)
100  if v = v_stab_r then
101    output om_r.installAbortStaet(b)

103 upon global call isCorrectViewChangeCertificate(V ⊆ Message, v_from ∈ ᵛℕ, v_to ∈ ᵛℕ) do
104   return |V| ≥ q ∧ (∀vc ∈ V ∃v_from^≤ :
        v_from^≤ ≤ v_from ∧ isCorrectViewCange(vc, ·, v_from^≤, v_to)

106 upon internal call isInOrderWindow(K ⊆ Message)
107   return checkpointBase(checkpointNumber(K)) ≤ input om_r.orderWindowBase()
```

```
109 upon internal call isInOrderWindow(V ⊆ Message, A ⊆ Message)
110   return checkpointBase(maxCheckpointNumber(V, A)) ≤ input om_r.orderWindowBase()

112 upon internal call maxCheckpointNumber(V ⊆ Message, A ⊆ Message) do
113   return max({checkpointNumber(K) |
          ⟨VIEW-CHANGE, ·, ·, ·, K, ...⟩... ∈ V ∨ ⟨NEW-VIEW-ACK, ·, ·, K, ·⟩...})

115 upon internal task newViewExpected()
116   with
117     ¬vstatus_r.isInStableView() ∧ l' = leader(v_cur_r) ∧ l' ≠ r
118     ∃V ⊆ vmsgs_r : isCorrectViewChangeCertificate(V, v_stab_r, v_cur_r)
119     ¬nv_timeout_r.isRunning()
120   do
121     nv_timeout_r.start(l')

123 upon internal call suspectsDesignatedLeader() do
124   return nv_timeout_r.isExpired()

126 upon internal task designatedLeaderSuspected()
127   with
128     ¬vstatus_r.isInStableView()∧ (suspectsDesignatedLeader() ∨ isLeaderSuspectedByQuorum())
129     ∃V ⊆ vmsgs_r : isCorrectViewChangeCertificate(V, v_stab_r, v_cur_r)∧ isInOrderWindow(V, ∅)
130   do
131     installAbortStates(V)
132     leaveViewFor(v_cur_r + 1)

134 upon internal task viewMissed(v' ∈ ᵛℕ)
135   with
136       ∃V ⊆ vmsgs_r ∃v : v' > v_cur_r ∧ isCorrectViewChangeCertificate(V, v_stab_r, v')
              ∧ isInOrderWindow(V, ∅)
137   do
138     installAbortStates(V)
139     leaveViewFor(v')

141 upon internal call installAbortStates(V ⊆ Message)
142   for all vc ∈ V do
143     vc ≔ ⟨VIEW-CHANGE, ·, v_from, ··· , b⟩...
144       if v_from = v_stab_r then
145         output om_r.installAbortState(b)

147 upon global call isCorrectNewViewCertificate(V ⊆ Message, A ⊆ Message, v_from ∈ ᵛℕ, v_to ∈ ᵛℕ)
      do
148   return isCorrectViewChangeCertificate(V, v_from, v_to)
149       ∧ (∀na ∈ A : isCorrectNewViewAck(na, ·, v_from))
150       ∧ |{r' | ⟨VIEW-CHANGE, r', v_from, ...⟩... ∈ V ∨ ⟨NEW-VIEW-ACK, r', ...⟩... ∈ A}| > f

152 upon internal task newLeaderViewReady()
153   with
154     ∃v' ≥ v_cur_r ∃V, A ⊆ vmsgs_r : r = leader(v')
155       ∧ isCorrectNewViewCertificate(V, A, ·, v')
156       ∧ isInOrderWindow(V, A)
157   do
158     establishView(v', V, A)

160 upon internal call establishView(v' ∈ ᵛℕ, V ⊆ Message, A ⊆ Message) do
161   if v' > v_cur_r then
162     output om_r.abortView(v')
```

```
163     v_cur_r := v'
164   K := input km_r.stableCheckpointCertificate()
165   B := allAbortStates(V, A)
166   P' := input om_r.createViewTransition(v', B)
167   nv := ⟨New-View, r, v', K, V, A, P'⟩_{τ(r,𝔑,v',−)}
168   invoke tss_r.createIndependentCertficate(𝔑, v', nv)
169   storeNewView(nv)
170   enterView(v', P')
171   output repgrpconn_r.send(nv)

173 upon global call isCorrectNewView(nv, v ∈ ᵛℕ, K ⊆ Message) do
174   return ∃V, A, P', o_stab, o_base:
175       nv = ⟨New-View, leader(v), v, K, V, P'⟩_{τ(l,𝔑,v,−)}
176       ∧ hasValidCertificate(nv)
177       ∧ isCorrectNewViewCertificate(V, A, ·, v)
178       ∧ isCorrectCheckpointCertificate(K, o_stab, ·)
179       ∧ o_stab ≥ maxCheckpointNumber(V, A)
180       ∧ o_base = checkpointBase(o_stab)
181       ∧ isCorrectViewTransition(P', v, o_base, allAbortStates(V, A))

183 upon input call receive(nv' = ⟨New-View, ...⟩_...) do
184   if isRelevantNewView(nv') ∧ isCorrectNewView(nv', ...) then
185     installNewView(nv')

187 upon internal call isRelevantNewView(nv' = ⟨New-View, ·, v', ...⟩_...) do
188   return v' > latestKnownView()

190 upon internal call storeNewView(nv' = ⟨New-View, ...⟩_...) do
191   vmsgs_r := vmsgs_r ∪ {nv'}
192   discardNewViews()

194 upon internal call discardNewViews() do
195   NV^cur := {nv_cur ∈ vmsgs_r | nv_cur = ⟨New-View, ·, v_cur, ...⟩_...
          ∧ (v_cur = v_stab_r ∨ v_cur = latestKnownView()}
196   vmsgs_r := vmsgs_r \ {nv ∈ vmsgs_r | nv = ⟨New-View, ...⟩_... ∧ nv ∉ NV^cur}
197   output repgrpconn_r.remove({nv | nv = ⟨New-View, ...⟩_... ∧ nv ∉ NV^cur})

199 upon internal call installNewView(nv' = ⟨New-View, l, v', K, V, P'⟩_...) do
200   storeNewView(nv')
201   for all vc ∈ V do
202     if isRelevantViewChange(vc) then
203       installViewChange(vc)
204   output km_r.installCheckpointCertificate(K)
205   for all ⟨View-Change, ..., K', ...⟩_... ∈ V
206     output km_r.installCheckpointCertificate(K')

208 upon internal call allAbortStates(V ⊆ Message, A ⊆ Message) do
209   return {b | ⟨View-Change, ..., b⟩_... ∈ V ∨ ⟨New-View-Ack, ..., b⟩_... ∈ A}

211 upon internal call latestKnownView() do
212   return max({v | ⟨New-View, ·, v, ...⟩_... ∈ vmsgs_r} ∪ {0})

214 upon internal task newFollowerViewReady(nv = ⟨New-View, l, v', K, ·, ·, P'⟩_...)
215   with
216     nv ∈ vmsgs_r ∧ v' = latestKnownView() ∧ v' > v_stab_r ∧ isInOrderWindow(K)
217   do
218     isbelated := v' < v_cur_r
```

```
219     if v' > v_cur_r then
220       output om_r.abortView(v')
221       v_cur_r := v'
222     enterView(v', P')
223     if isbelated then
224       sendNewViewAck()

226 upon internal call sendNewViewAck() do
227   K := input km_r.stableCheckpointCertificate()
228   b := input om_r.createAbortState(v_cur_r)
229   na := ⟨New-View-Ack, r, v_cur_r, K, b⟩_τ(r,𝔐,0,0)
230   invoke tss_r.createContinuingCertficate(𝔐, 0, na)
231   storeNewViewAck(na)
232   output repgrpconn_r.send(na)

234 upon global call isCorrectNewViewAck(na, r' ∈ ℛ, v ∈ ᵛℕ) do
235   return ∃K ⊆ Message ∃b ∈ AbortState ∃o_base, o_stab ∈ ᵒℕ:
236       na = ⟨New-View-Ack, r', v, K, b⟩_τ(r,𝔐,0,0)
237       ∧ hasValidCertificate(na)
238       ∧ isCorrectCheckpointCertificate(K, o_stab, ·)
239       ∧ o_base = checkpointBase(o_stab)
240       ∧ isCorrectAbortState(b, v_from, v_to, o_base)

242 upon input call receive(na' = ⟨New-View-Ack, ...⟩...) do
243   if isRelevantNewViewAck(na') ∧ isCorrectNewViewAck(na', ···) then
244     installNewViewAck(na')

246 upon internal call isRelevantNewViewAck(na' = ⟨New-View-Ack, r', v', ...⟩...) do
247   return v' ≥ v_stab_r ∧ (∀na ∈ vmsgs_r : na = ⟨New-View-Ack, r', v, ...⟩... → v' > v)

249 upon internal call storeNewViewAck(na' = ⟨New-View-Ack, ...⟩...) do
250   vmsgs_r := vmsgs_r ∪ {na}
251   discardNewViewAcks()

253 upon internal call discardNewViewAcks() do
254   NA^max := {na_max ∈ vmsgs_r | na_max = ⟨New-View-Ack, r', v_max, ...⟩...
255 ∧ v_max ≥ v_stab_r
256 ∧ (∀na ∈ vmsgs_r : na = ⟨New-View-Ack, r', v, ...⟩... → v ≤ v_max}
257   vmsgs_r := vmsgs_r \ {na ∈ vmsgs_r | na = ⟨New-View-Ack, ...⟩... ∧ na ∉ NA^max}
258   output repgrpconn_r.remove({na | na = ⟨New-View-Ack, ...⟩... ∧ na ∉ NA^max})

260 upon internal call installNewViewAck(na' = ⟨New-View-Ack, ·, v', K, b⟩...) do
261   storeNewViewAck(na')
262   abortStateReceived(v', K, b)

264 upon internal call enterView(v' ∈ ᵛℕ, P' ⊆ Message) do
265   vstatus_r.enterView(v')
266   discardViewChanges()
267   discardNewViews()
268   discardNewViewAcks()
269   if vstatus_r.isInStableView() then
270     if nv_timeout_r.isRunning() then
271       nv_timeout_r.stop()
272     output sm_r.stopStateRequestTask()
273   output om_r.enterView(v', P')
274   output cm_r.enterView(v')
```

```
upon output call createStateRequestContent() do
  o_comm := input om_r.lastCommitted()
  o_stab := input km_r.stableCheckpointNumber()
  return ⟨latestKnownView(), v_cur_r, o_stab, o_comm⟩

upon global call isCorrectStateRequestContent(sr) do
  return sr = ⟨v_latest ∈ ᵛℕ, v_cur ∈ ᵛℕ, o_stab ∈ ᵒℕ, o_comm ∈ ᵒℕ⟩ ∧ o_stab ≤ o_comm

upon output call obtainState(⟨v_latest, v_cur, o_stab, o_comm⟩) do
  if v_latest > v_stab_r ∨ v_stab_r ≠ latestKnownView()
    return ∅
  if v_latest < v_stab_r then
    nv := stableViewCertificate()
    I := input om_r.obtainOrderingState( input om_r.orderWindowBase())
  else
    nv := ∅
    I := input om_r.obtainOrderingState(o_comm)
  V := {vc ∈ vmsgs_r | vc = ⟨VIEW-CHANGE, ..., v_to, ·⟩...
        ∧ (v_to > v_latest ≥ v_cur ∨ v_to ≥ v_cur > v_latest)}
  A := {na ∈ vmsgs_r | na = ⟨NEW-VIEW-ACK, ·, v, ...⟩... ∧ v ≥ v_latest}
  ks := input km_r.obtainCheckpointingState(o_stab, o_comm),
  return if nv ≠ ∅ ∨ |V| > 0 ∨ |A| > 0 ∨ ks ≠ ∅ ∨ |I| > 0
            then ⟨v_stab_r, nv, V, A, ks, I⟩ else ∅

upon global call isCorrectState(state)
  return state = ⟨v, nv, V, A, ks, I⟩ ∧ (∃K', o_base, o_stab:
      ∧ (nv = ∅ ∨ isCorrectNewView(nv, v, K', ...) ∧ o_stab ≥ checkpointNumber(K'))
      ∧ (∀vc ∈ V ∃v_to: isCorrectViewChange(vc, ·, ·, v_to) ∧ v_to > v)
      ∧ (∀na ∈ A ∃v': isCorrectNewViewAck(na, ·, v') ∧ v' ≥ v)
      ∧ o_base = checkpointBase(o_stab)
      ∧ isCorrectCheckpointingState(ks, o_stab)
      ∧ isCorrectOrderingState(I, v, o_base))

upon input call installState(⟨v, nv, V, A, ks, I⟩)
  output km_r.installCheckpointingState(ks)
  if isRelevantNewView(nv) then
    installNewView(nv)
  check newFollowerViewReady()
  if v = v_stab_r then
    output om_r.installOrderingState(I)
  for all vc ∈ V do
    if isRelevantViewChange(vc) then
      installViewChange(vc)
  for all na ∈ A do
    if isRelevantNewViewAck(na) then
      installNewViewAck(na)
```

4.10 State Transfer Protocol

Listing 20: Specification of the state-transfer protocol.

```
1  module Replica_r::StateTransfer

3  uses vm_r ∈ Replica_r::ViewChange

5  let state_period_r ∈ ℕ

7  upon init() do
8    state_timer_r ≔ Timer()

10 upon input call startStateRequestTask() do
11   output state_timer_r.schedule(state_period_r)

13 upon input call stopStateRequestTask() do
14   output state_timer_r.cancel()

16 upon internal task periodicallyRequestState()
17   with
18     input state_timer_r.isExpired()
19   do
20     requestState()
21     state_timer_r.schedule(state_period_r)

23 upon input call requestState() do
24   src ≔ input vm_r.createStateRequestContent()
25   sr ≔ ⟨STATE-REQUEST, r, src⟩_τ(r,𝔐,0,0)
26   invoke tss_r.createConsecutiveCertificate(𝔐, 0, sr)
27   output repgrpconn_r.remove({m | m = ⟨STATE-REQUEST, ...⟩...})
28   output repgrpconn_r.send(sr)

30 upon global call isCorrectStateRequest(sr, r′ ∈ ℛ, src) do
31   return sr = ⟨STATE-REQUEST, r′, src⟩_τ(r′,𝔐,0,0)
          ∧ hasValidCertificate(sr) ∧ isCorrectStateRequestContent(src)

33 upon input call receive(rs = ⟨STATE-REQUEST, r′, src⟩...) do
34   if isCorrectStateRequest(rs, r′, src) then
35     state ≔ input vm_r.obtainState(src)
36     if state ≠ ∅
37       sendState(r′, state)

39 upon internal call sendState(r′ ∈ ℛ, state) do
40   sm ≔ ⟨STATE, r, state⟩_τ(r,𝔐,0,0)
41   invoke tss_r.createConsecutiveCertificate(𝔐, 0, sm)
42   output repconns_r(r′).remove({m | m = ⟨STATE, ...⟩...})
43   output repconns_r(r′).send(sm)

45 upon global call isCorrectStateMessage(sm, r′ ∈ ℛ, state) do
46   return sm = ⟨STATE, r′, state⟩_τ(r′,𝔐,0,0) ∧ hasValidCertificate(sm)
47        ∧ isCorrectState(state)

49 upon input call receive(sm = ⟨STATE, r′, state⟩...) do
50   if isCorrectStateMessage(sm, r′, state) then
51     output vm_r.installState(state)
```

References

[1] D. J. Armstrong. The quarks of object-oriented development. *Communications of the ACM*, 49(2):123–128, 2006.

[2] J. Behl, T. Distler, and R. Kapitza. Hybster — A highly parallelizable protocol for hybrid fault-tolerant service replication. `http://publikationsserver.tu-braunschweig.de/get/64440`.

[3] J. Behl, T. Distler, and R. Kapitza. Hybrids on steroids: SGX-based high performance BFT. In *Proceedings of the 12th European Conference on Computer Systems (EuroSys '17)*, 2017.

[4] C. Cachin, R. Guerraoui, and L. Rodrigues. *Introduction to Reliable and Secure Distributed Programming.* Springer Publishing Company, Incorporated, 2nd edition, 2011.

[5] M. Castro. *Practical Byzantine Fault-Tolerance.* PhD thesis, MIT, 2001.

[6] M. Castro and B. Liskov. A correctness proof for a practical Byzantine-fault-tolerant replication algorithm. Technical report, Cambridge, MA, USA, 1999.

[7] M. Castro and B. Liskov. Practical Byzantine fault tolerance. In *Proceedings of the 3rd Symposium on Operating Systems Design and Implementation (OSDI '99)*, pages 173–186, 1999.

[8] M. Castro and B. Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, Nov. 2002.

[9] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, Apr. 1988.

[10] M. P. Herlihy and J. M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[11] N. A. Lynch. *Distributed Algorithms.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[12] F. J. Meyer and D. K. Pradhan. Consensus with dual failure modes. In *Seventeenth International Symposium on Fault Tolerant Computing*, volume 2, pages 214–222, Apr. 1987.

[13] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Computing Surveys*, 22(4):299–319, 1990.

[14] P. Thambidurai and Y. keun Park. Interactive consistency with multiple failure modes. In *Proceedings of the 7th IEEE International Symposium on Reliable Distributed Systems (SRDS '88)*, pages 93–100, Oct. 1988.